

# Introduction to Concurrent Programming

**Lesson 1** of TDA384/DIT391

Principles of Concurrent Programming

Nir Piterman and Gerardo Schneider

Chalmers University of Technology | University of Gothenburg



UNIVERSITY OF  
GOTHENBURG



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY

# Today's menu

- A motivating example
- Why concurrency?
- Basic terminology and abstractions
- Java threads
- Traces

# A Motivating Example

# As simple as counting to two

We illustrate the **challenges** introduced by concurrent programming on a simple example: a **counter** modeled by a Java class

- First, we write a traditional, **sequential** version
- Then, we introduce **concurrency** and...run into **trouble!**

# Sequential counter

```
public class Counter {  
    private int counter = 0;  
  
    // increment counter by one  
    public void run() {  
        int cnt = counter;  
        counter = cnt + 1;  
    }  
  
    // current value of counter  
    public int counter() {  
        return counter;  
    }  
}
```

```
public class SequentialCount {  
    public static  
    void main(String[] args) {  
        Counter counter = new Counter();  
        counter.run(); // increment once  
        counter.run(); // increment  
        // print final value of counter  
        System.out.println(  
            counter.counter());  
    }  
}
```

- What is printed by running: `java SequentialCount`?
- May the printed value change in different reruns?

# Modeling sequential computation

```
5  public void run() {  
6      int cnt = counter;  
7      counter = cnt + 1;  
8  }
```

```
counter.run(); // first call: steps 1-3  
counter.run(); // second call: steps 4-6
```

#	LOCAL STATE	OBJECT STATE
1	pc: 6	cnt: ⊥ counter: 0
2	pc: 7	cnt: 0 counter: 0
3	pc: 8	cnt: 0 counter: 1
4	pc: 6	cnt: ⊥ counter: 1
5	pc: 7	cnt: 1 counter: 1
6	pc: 8	cnt: 1 counter: 2
7	done	counter: 2

# Adding concurrency

Now, we revisit the example by introducing **concurrency**:

Each of the two calls to method `run` can be executed in **parallel**

- In Java, this is achieved by using **threads**
- Do not worry about the details of the syntax for now, we will explain it later

The idea is just that:

- There are two independent execution units (**threads**) `t` and `u`
- Each execution unit executes `run` on the **same** `counter` object
- We have **no control** over the **order of execution** of `t` and `u`

# Concurrent counter

```
public class CCounter
    extends Counter
    implements Runnable
{
    // threads
    // will execute
    // run()
}
```

```
public class ConcurrentCount {
    public static void main(String[] args) {
        CCounter counter = new CCounter();
        // threads t and u, sharing counter
        Thread t = new Thread(counter);
        Thread u = new Thread(counter);
        t.start(); // increment once
        u.start(); // increment twice
        try { // wait for t and u to terminate
            t.join(); u.join();
        } catch (InterruptedException e) {
            System.out.println("Interrupted!");
        } // print final value of counter
        System.out.println(counter.counter());
    }
}
```

- What is printed by running: `java ConcurrentCount`?
- May the printed value change in different reruns?

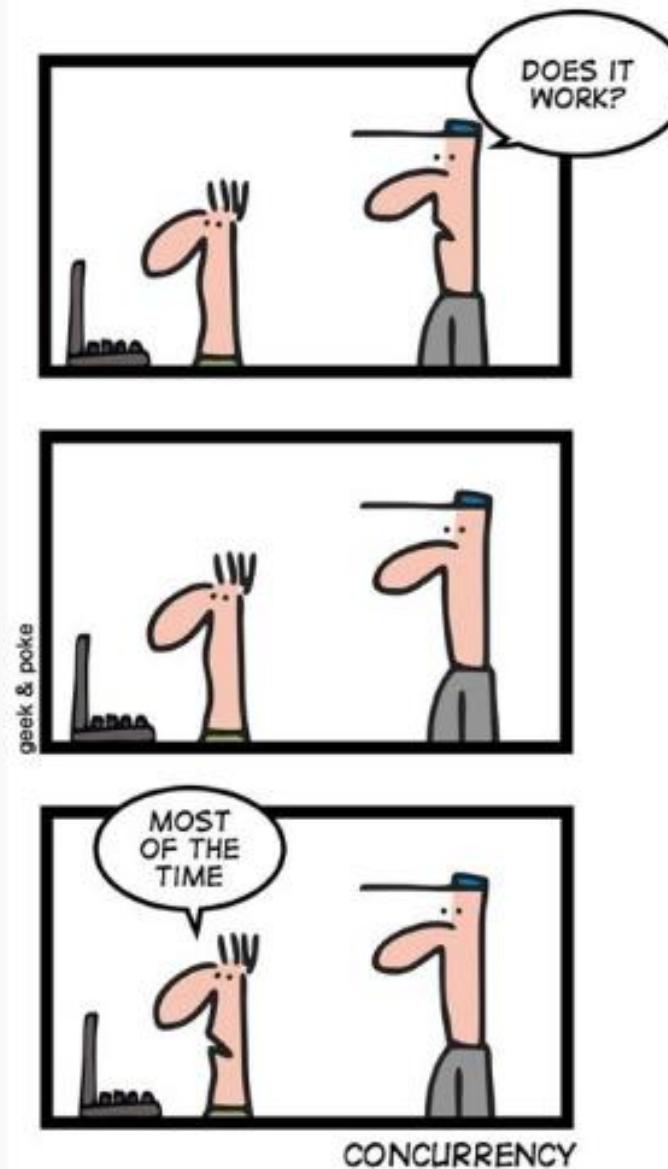
# What?!

```
$ javac Counter.java CCounter.java ConcurrentCount.java
$ java ConcurrentCount.java
2
$ java ConcurrentCount.java
2
...
$ java ConcurrentCount.java
1←
$ java ConcurrentCount.java
2
```

The concurrent version of counter occasionally prints 1 instead of the expected 2

- It seems to do so **unpredictably**

Welcome to concurrent programming!



# Why concurrency?

# Reasons for using concurrency

Why do we need concurrent programming in the first place?

- **Abstraction:**
  - Separating different tasks, without worrying about when to execute them (Ex: download files from two different websites)
- **Responsiveness:**
  - Providing a responsive user interface, with different tasks executing independently (Ex: browse the slides while downloading your email)
- **Performance:**
  - Splitting complex tasks in multiple units, and assign each unit to a different processor (Ex: compute all prime numbers up to 1 billion)

# Concurrency vs. parallelism

Principles of concurrent programming

vs.

Principler för parallel programming

Huh?

# Concurrency vs. parallelism

We will mostly use **concurrency** and **parallelism** as synonyms

However, they refer to similar but different concepts:

- **Concurrency**: nondeterministic composition of independently executing units (**logical parallelism**)
- **Parallelism**: efficient execution of fractions of a complex task on multiple processing units (**physical parallelism**)
- You can have **concurrency without physical parallelism**: operating systems running on single-processor single-core systems
- **Parallelism** is mainly about **speeding up** computations by taking advantage of redundant hardware

# Concurrency vs. parallelism

Ideal situation



Photo: Summer Olympics 2016, Sander van Ginkel.

# Concurrency vs. parallelism

More common situation



Photos: World Cup Nordic '07, Tomoyoshi Noguchi – Vasaloppet '06, Steven Hale.

# Concurrency vs. parallelism

Real world situation



Photo: Daniel Mott 2009



Photo: Wolfgangus Mozart 2010

## Challenges:

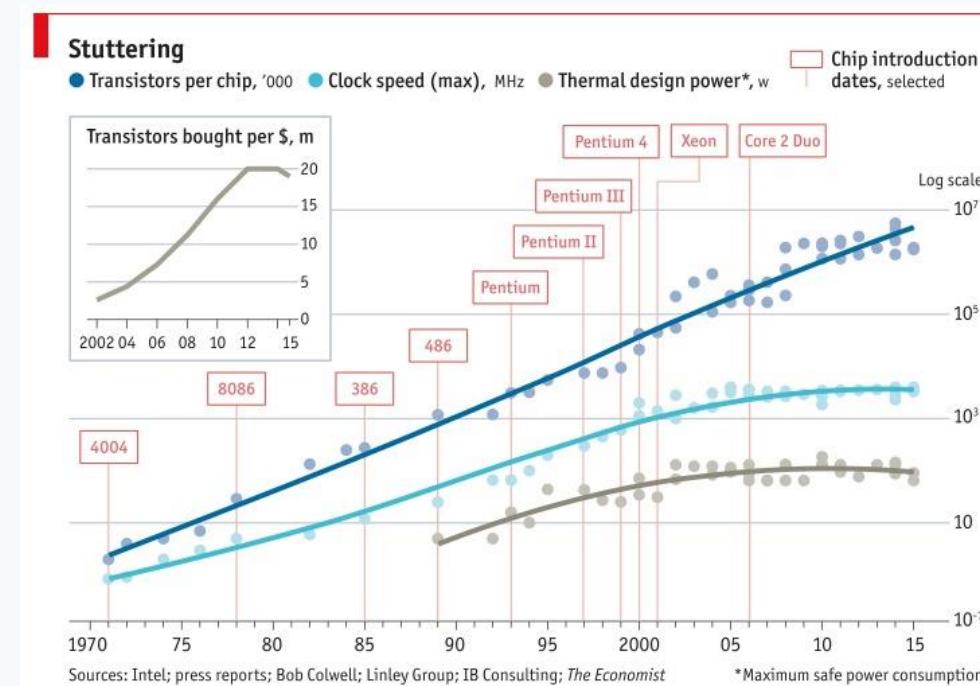
- *Concurrency*: Everyone gets to do their laundry (**fairness**)  
Machines are operated by at most one user (**mutual exclusion**)
- *Parallelism*: Distribute load evenly over machines/rooms (**load balancing**)

**Solutions:** schedules, locks, signs/indicators...

# Moore's law and its end (?)

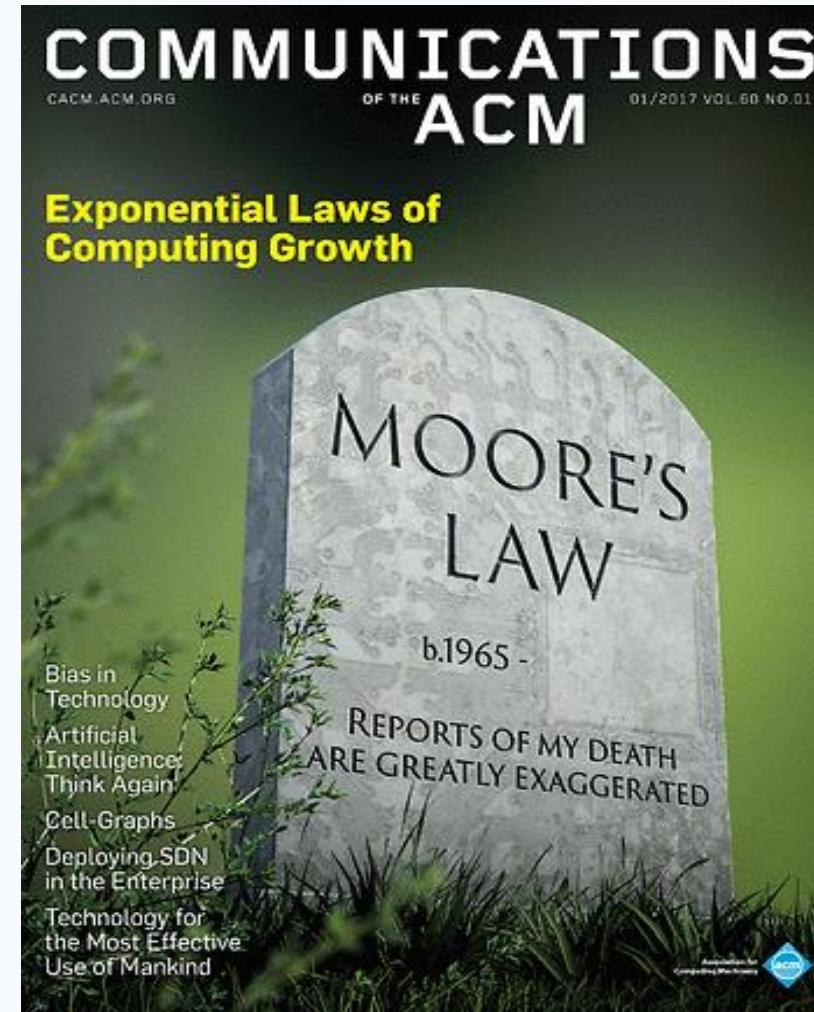
The spectacular advance of computing in the last 60+ years has been driven by **Moore's law** (1965)

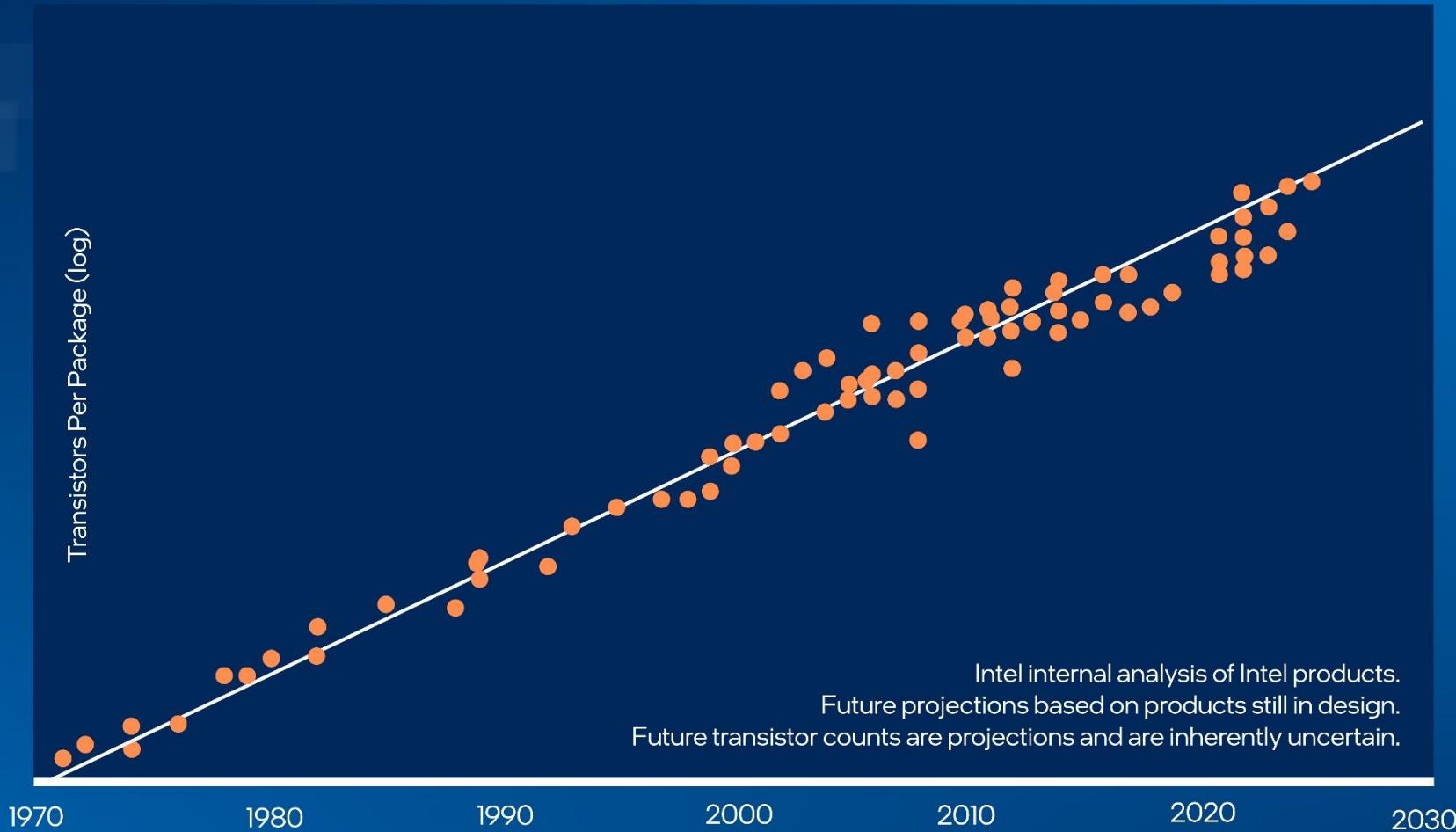
1975: The density of transistors in integrated circuits  
**doubles approximately every 2 years**



Later updated:  
**Doubling every  
 18 months**  
 (instead of 2 years)

# Moore's Law in January 2017





Aspiring to  
**1 Trillion**  
transistors in 2030

- RibbonFET
- PowerVia
- High NA
- 2.5D/3D packaging

## Opinion

- February 16, 2022
- [Download a PDF version of this editorial](#)
- [Contact Intel PR](#)

[More Manufacturing News](#)

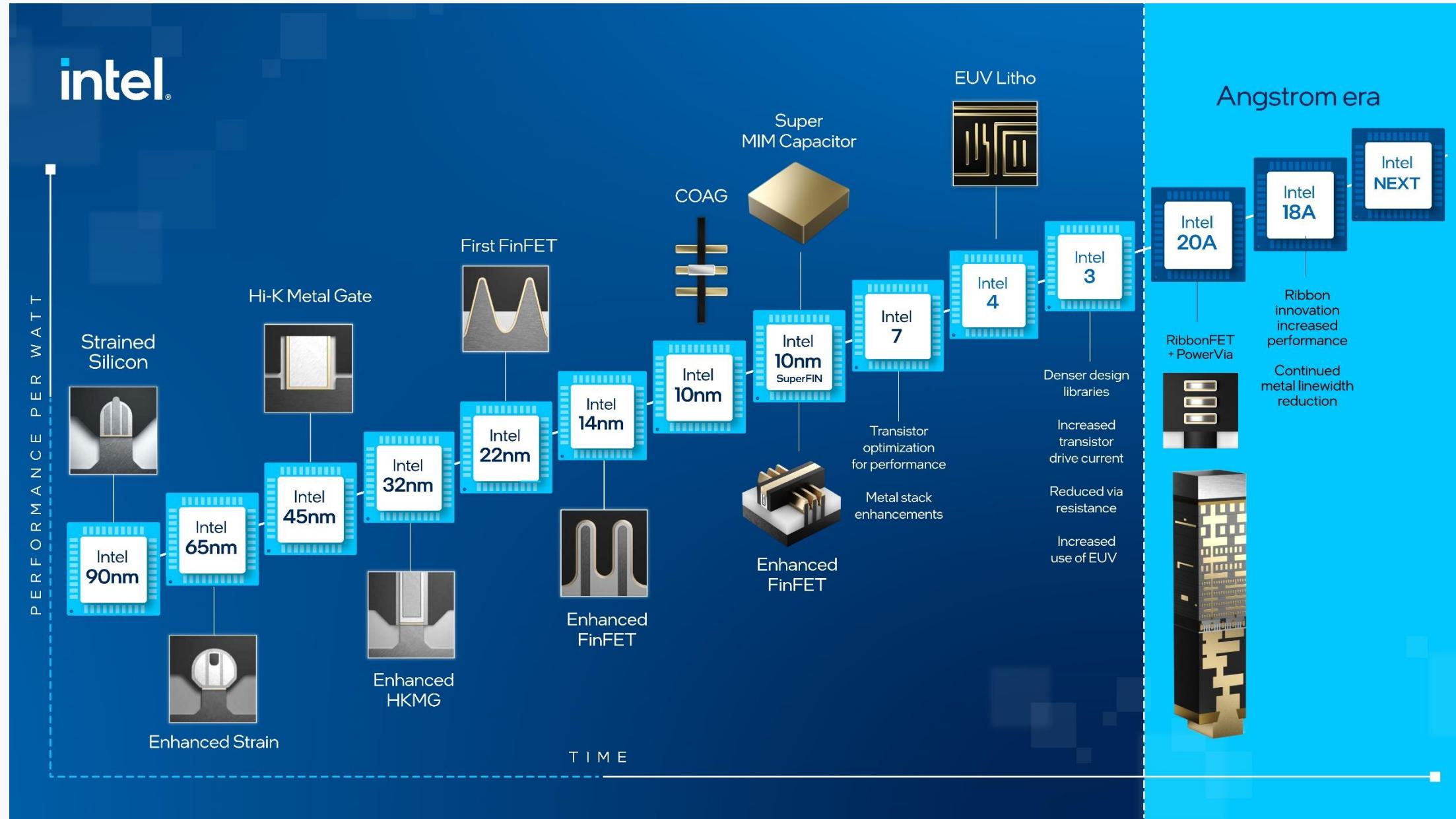
## Executive Summary

- Intel has a rich history of foundational process innovations in pursuit of Moore's Law.
- Advanced packaging gives architects and designers new tools in their pursuit of Moore's Law.
- Intel has a full pipeline of research that gives us the confidence of maintaining Moore's Law.
- All considered, numerous options are available to designers and architects in their continued mission to deliver Moore's Law



**By Dr. Ann Kelleher**

*Executive Vice President and General Manager of Technology Development*



# Concurrency everywhere

Physical restrictions force to change from increasing processing speed to having multiple processing having a major impact on the practice of **programming**:

- **Before:** CPU **speed** increases without significant architectural changes
  - Concurrent programming was a **niche skill** (for operating systems, databases, high-performance computing)
  - Program **as usual** and wait for your program to run faster
- **Now:** CPU speed remains the same, but **number of cores** increases
  - Concurrent programming is **pervasive**
  - Program **with concurrency** in mind, otherwise your programs remain slow

Very different systems all require concurrent programming:

- desktop PCs,
- smart phones,
- video-games consoles,
- embedded systems,
- the Raspberry Pi,
- cloud computing, ...

# Amdahl's law: Concurrency is no free lunch

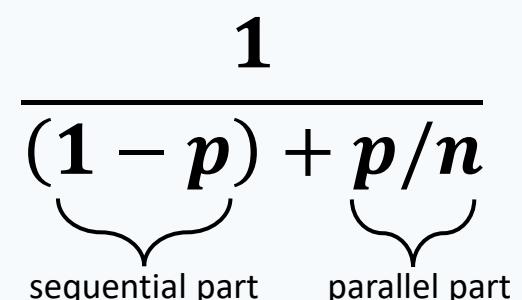
We have  $n$  processors that can run in parallel

How much speedup can we achieve?

$$\text{speedup} = \frac{\text{sequential execution time}}{\text{parallel execution time}}$$

Amdahl's law shows that the impact of introducing parallelism is limited by the fraction  $p$  of a program that can be parallelized:

$$\text{maximum speedup} = \frac{1}{(1-p) + p/n}$$



# Amdahl's law: Examples

$$\text{maximum speedup} = \frac{1}{(1-p) + p/n}$$

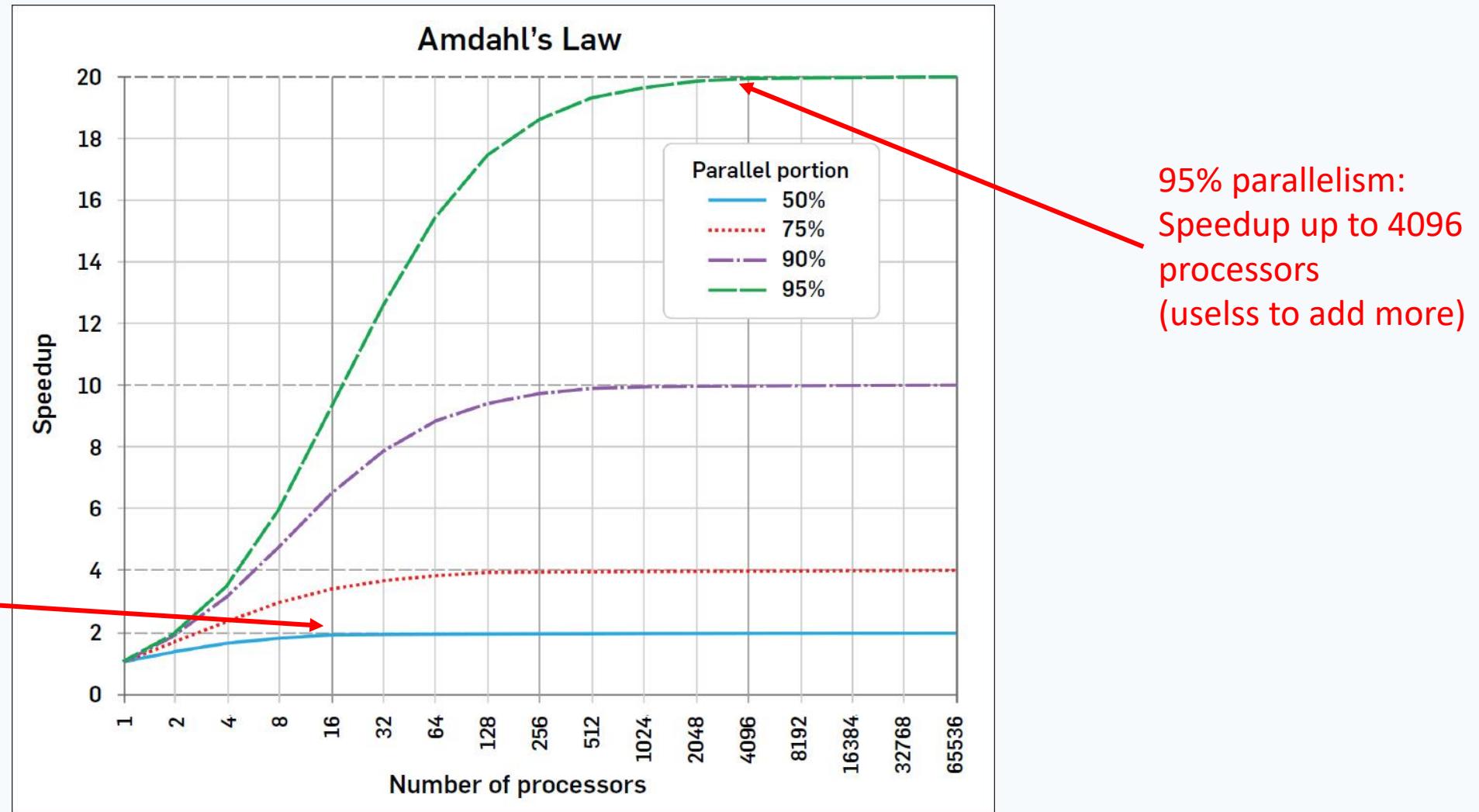
With  $n=10$  processors, how close can we get to a 10x speedup?

% SEQUENTIAL	% PARALLEL	MAX SPEEDUP
20%	80%	3.57
10%	90%	5.26
1%	99%	9.17

With  $n=100$  processors, how close can we get to a 100x speedup?

% SEQUENTIAL	% PARALLEL	MAX SPEEDUP
20%	80%	4.81
10%	90%	9.17
1%	99%	50.25

# Amdahl's law: Examples



Source: Communications of the ACM, Dec. 2017

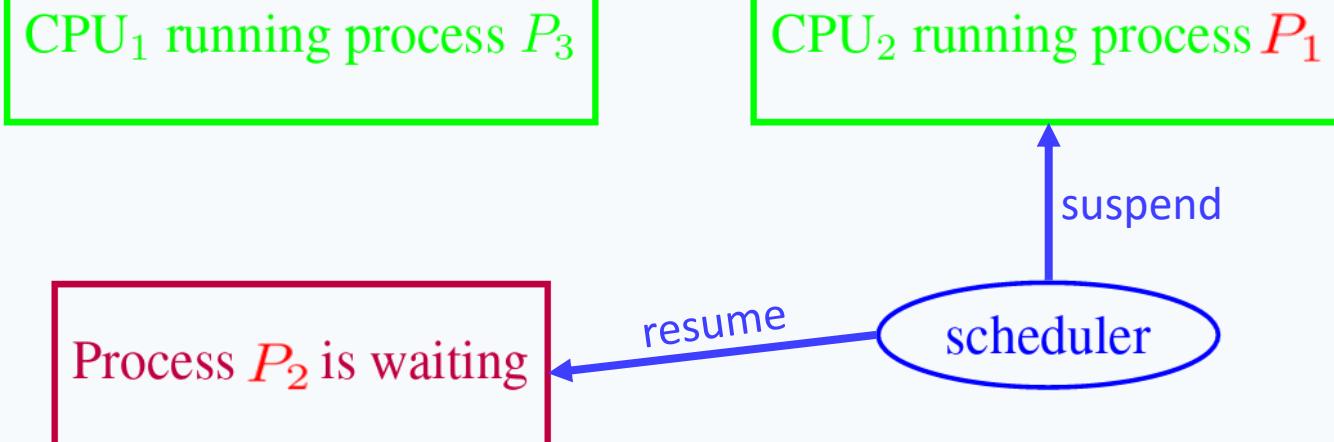
# Basic terminology and abstractions

# Processes

A **process** is an **independent unit of execution** – the abstraction of a running sequential program:

- identifier
- program counter (PC)
- memory space

The runtime/operating system **schedules** processes for execution on the available processors:



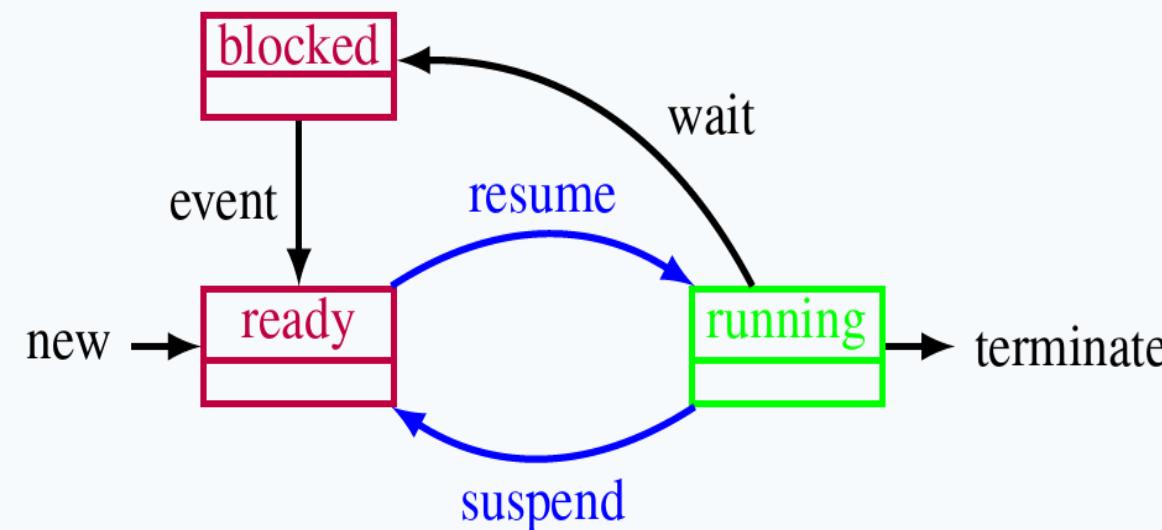
# Process states

The **scheduler** is the system unit in charge of setting **process states**:

**Ready:** ready to be executed, but not allocated to any CPU

**Blocked:** waiting for an event to happen

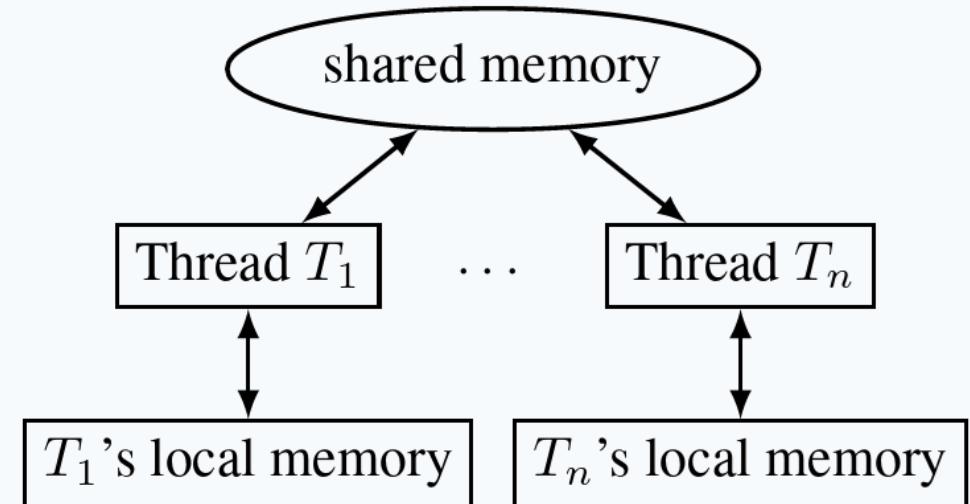
**Running:** running on some CPU



# Threads

A **thread** is a **lightweight process** – an independent unit of execution in the same program space:

- identifier
- program counter (PC)
- memory
  - **local** memory, separate for each thread
  - **global** memory, **shared** with other threads



In practice, the difference between processes and threads is fuzzy and implementation dependent. In our course:

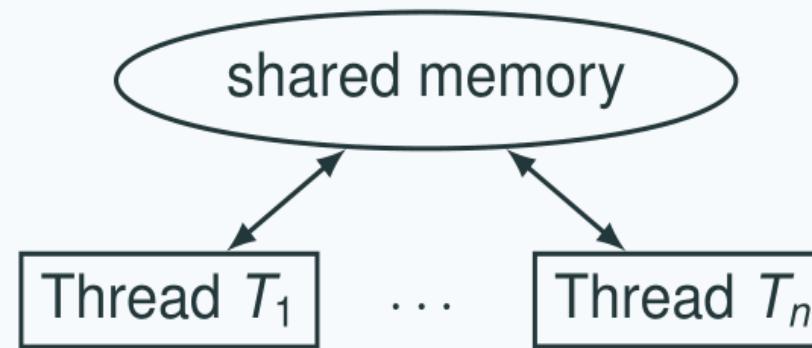
**Processes**: executing units that do **not share memory** (in Erlang)

**Threads**: executing units that **share memory** (in Java)

# Shared memory vs. message passing

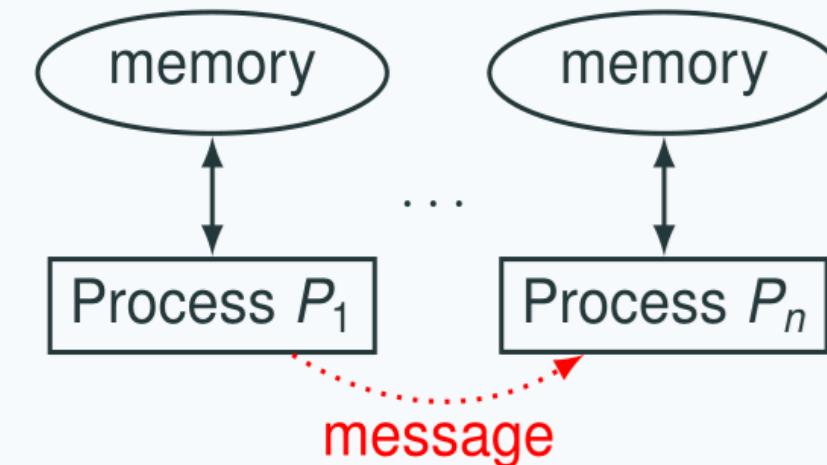
## Shared memory models:

- communication by writing to **shared memory**
- e.g., multi-core systems



## Distributed memory models:

- communication by **message passing**
- e.g., distributed systems

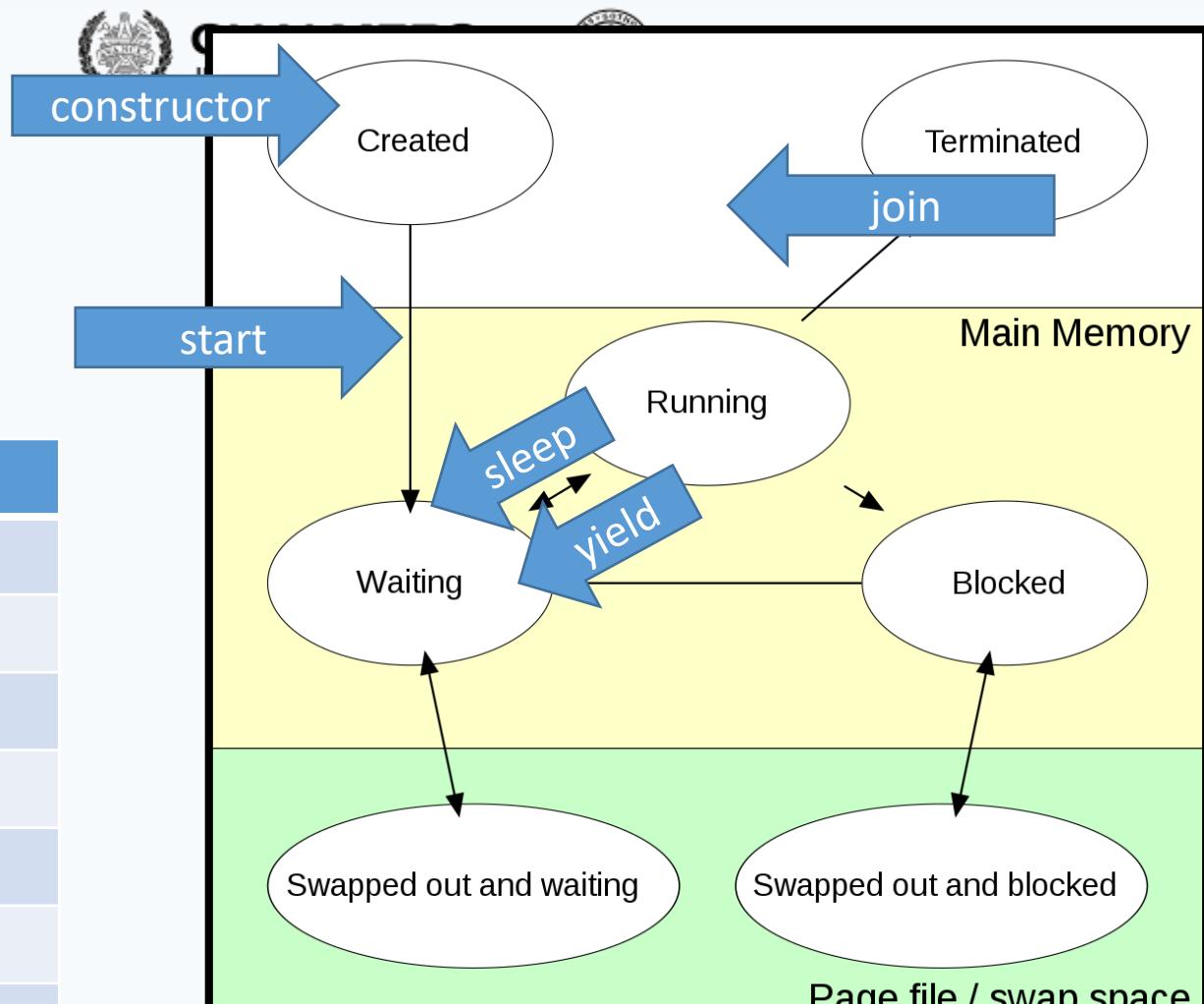


# Java threads

# Creating Threads

- What does a thread need to do?

Method	
start()	Start a thread by calling run() method
run()	Entry point for a thread
join()	Wait for a thread to end
isAlive()	Checks if thread is still running or not
setName()	
getName()	
getPriority()	



[https://en.wikipedia.org/wiki/Process\\_state](https://en.wikipedia.org/wiki/Process_state)

# Extend Thread

```
class MyThread extends Thread
{
    public void run()
    {
        System.out.println("concurrent thread started running..");
    }
}

class MyThreadDemo
{
    public static void main(String args[])
    {
        MyThread mt = new MyThread();
        mt.start();
    }
}
```

# Extend?

## Hierarchy: Animals

- Animal
  - Mammal
    - Canine
      - Dog
      - Wolf
    - Feline
      - Cat
  - Fish
    - Tuna
    - Shark
  - Reptile
    - Crocodile
    - Iguana

## Object - Bank Account

- Accounts have certain data and operations
  - Regardless of whether checking, savings, etc.
- Data
  - account number
  - balance
  - owner
- Operations
  - open
  - close
  - get balance
  - deposit
  - withdraw

## Kinds of Bank Accounts

- Account
  - **Checking**
    - Monthly fees
    - Minimum balance.
  - **Savings**
    - Interest rate
- Each type shares some data and operations of "account", and has some data and operations of its own.

Adv

# Implement Runnable

- Java does not support multiple inheritance
- If you need your class to inherit

```
class MyThread implements Runnable
{
    public void run()
    {
        System.out.println("concurrent thread started running..");
    }
}

class MyThreadDemo
{
    public static void main(String args[])
    {
        MyThread mt = new MyThread();
        Thread t = new Thread(mt);
        t.start();
    }
}
```

# Java threads

Two ways to build **multi-threaded** programs in Java:

- inherit from class Thread, override method run
- implement interface Runnable, implement method run

```
public class Ccounter
    extends Counter
    implements Runnable
{
    // thread's computation:
    public void run() {
        int cnt = counter;
        counter = cnt + 1;
    }
}
```

```
CCounter c = new CCounter();
```

```
Thread t = new Thread(c);
```

```
Thread u = new Thread(c);
```

```
t.start();
```

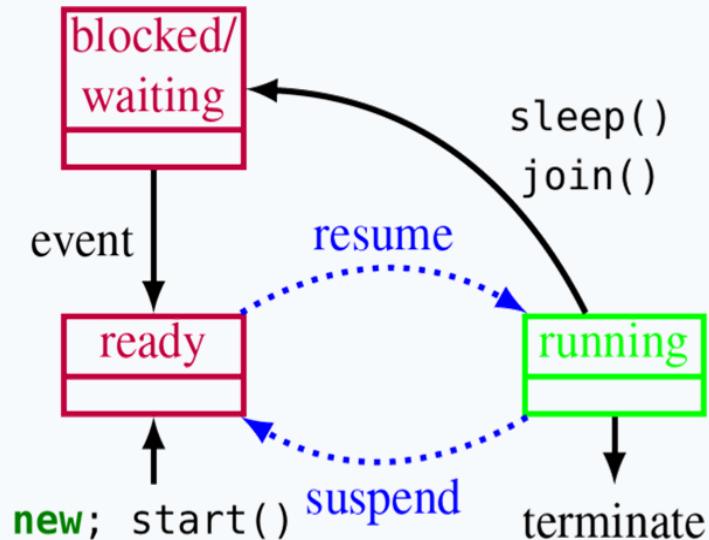
```
u.start();
```

Cannot use  
Thread class!

It inherits from  
Counter

So,  
can only use  
second method

# States of a Java thread

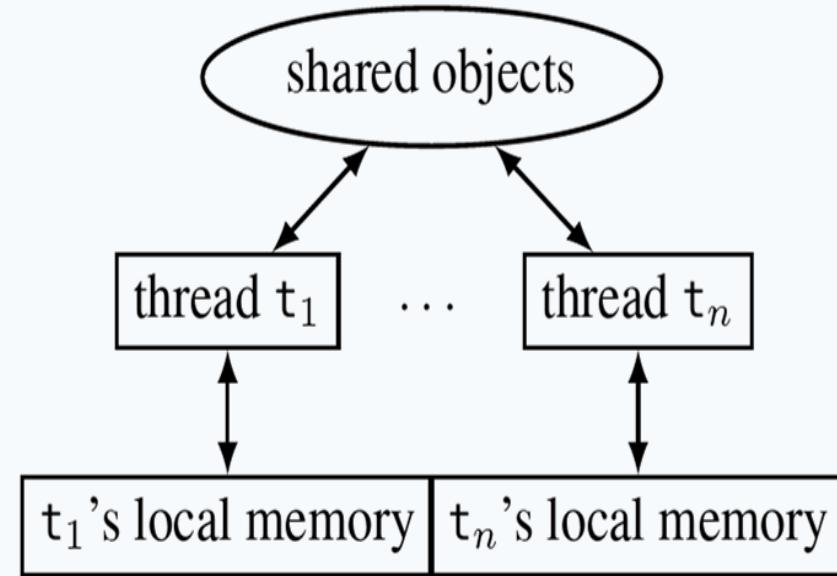


For a Thread object  $t$ :

- $t.start()$ : mark the thread  $t$  ready for execution
- $\text{Thread.sleep}(n)$ : block the **current thread** for  $n$  milliseconds (correct timing depends on JVM implementation)
- $t.join()$ : block the **current thread** until  $t$  terminates

Resuming and suspending is done by the **JVM scheduler**, outside the program's control

# Thread execution model



Shared vs. thread-local memory:

- **Shared objects**: the objects on which the thread operates, and all reachable objects
- **Local memory**: local variables, and special *thread-local* attributes

Threads proceed **asynchronously**, so they have to **coordinate** with other threads accessing the same shared objects

# One possible execution of the concurrent counter

```
1: public class CCounter implements Runnable {  
2:     int counter = 0;          // shared object state  
3:  
4:     // thread's computation:  
5:     public void run() {  
6:         int cnt = counter;  
7:         counter = cnt + 1;  
8:     } }
```

#	t's LOCAL	u's LOCAL	SHARED
1	pc <sub>t</sub> : 6 cnt <sub>t</sub> : ⊥	pc <sub>u</sub> : 6 cnt <sub>u</sub> : ⊥	counter: 0
2	pc <sub>t</sub> : 7 cnt <sub>t</sub> : 0	pc <sub>u</sub> : 6 cnt <sub>u</sub> : ⊥	counter: 0
3	pc <sub>t</sub> : 8 cnt <sub>t</sub> : 0	pc <sub>u</sub> : 6 cnt <sub>u</sub> : ⊥	counter: 1
4	done	pc <sub>u</sub> : 6 cnt <sub>u</sub> : ⊥	counter: 1
5	done	pc <sub>u</sub> : 7 cnt <sub>u</sub> : 1	counter: 1
6	done	pc <sub>u</sub> : 8 cnt <sub>u</sub> : 1	counter: 2
7	done	done	counter: 2

# One alternative execution of the concurrent counter

```

1: public class CCounter implements Runnable {
2:     int counter = 0;           // shared object state
3:
4:     // thread's computation:
5:     public void run() {
6:         int cnt = counter;
7:         counter = cnt + 1;
8:     }
  
```

#	t's LOCAL	u's LOCAL	SHARED
1	pc <sub>t</sub> : 6 cnt <sub>t</sub> : ⊥	pc <sub>u</sub> : 6 cnt <sub>u</sub> : ⊥	counter: 0
2	pc <sub>t</sub> : 7 cnt <sub>t</sub> : 0	pc <sub>u</sub> : 6 cnt <sub>u</sub> : ⊥	counter: 0
3	pc <sub>t</sub> : 7 cnt <sub>t</sub> : 0	pc <sub>u</sub> : 7 cnt <sub>u</sub> : 0	counter: 0
4	pc <sub>t</sub> : 7 cnt <sub>t</sub> : 0	pc <sub>u</sub> : 8 cnt <sub>u</sub> : 0	counter: 1
5	pc <sub>t</sub> : 8 cnt <sub>t</sub> : 0	done	counter: 1
6	done	done	counter: 1

# Traces

# Traces

#	t's LOCAL	u's LOCAL	SHARED
1	pc <sub>t</sub> : 6 cnt <sub>t</sub> : ⊥	pc <sub>u</sub> : 6 cnt <sub>u</sub> : ⊥	counter: 0
2	pc <sub>t</sub> : 7 cnt <sub>t</sub> : 0	pc <sub>u</sub> : 6 cnt <sub>u</sub> : ⊥	counter: 0
3	pc <sub>t</sub> : 7 cnt <sub>t</sub> : 0	pc <sub>u</sub> : 7 cnt <sub>u</sub> : 0	counter: 0
4	pc <sub>t</sub> : 7 cnt <sub>t</sub> : 0	pc <sub>u</sub> : 8 cnt <sub>u</sub> : 0	counter: 1
5	pc <sub>t</sub> : 8 cnt <sub>t</sub> : 0	done	counter: 1
6	done	done	counter: 1

The sequence of states gives an execution **trace** of the concurrent program

A trace is an **abstraction** of concrete executions:

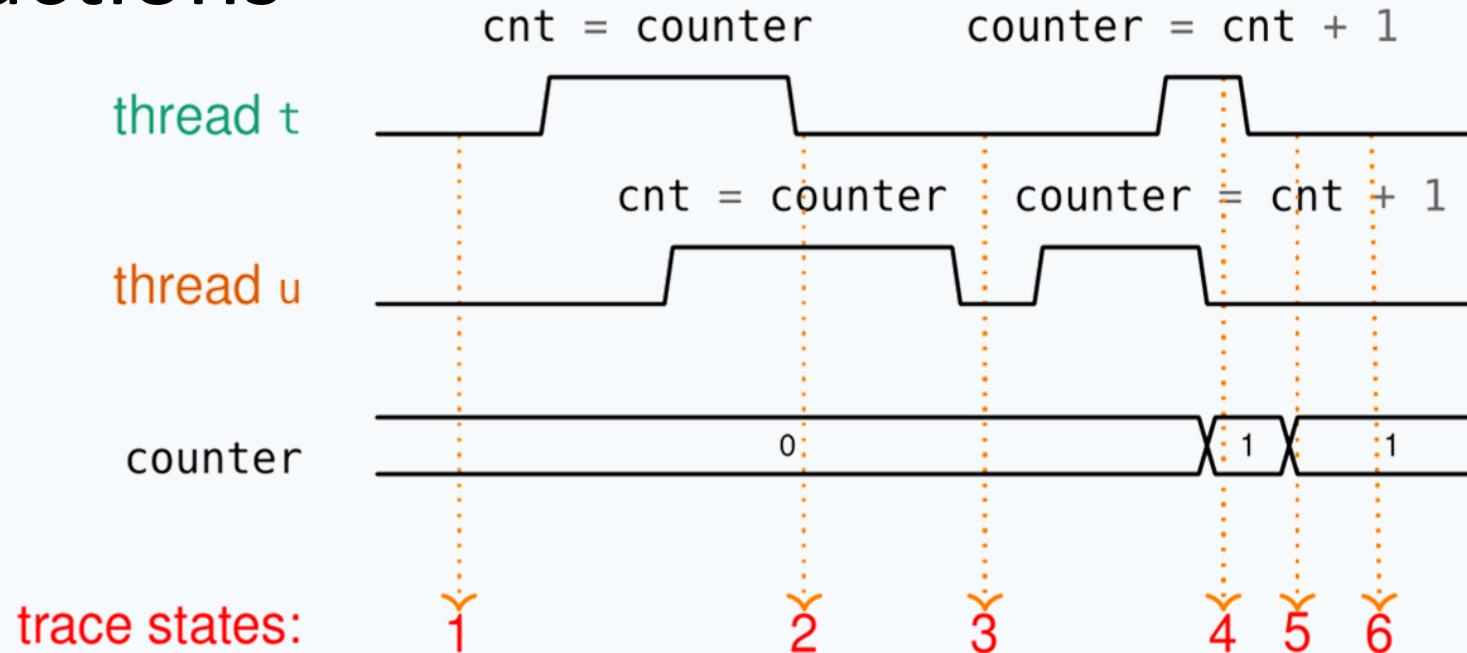
- atomic/linearized
- complete
- interleaved

Another trace  
A different  
interleaving



#	t's LOCAL	u's LOCAL	SHARED
1	pc <sub>t</sub> : 6 cnt <sub>t</sub> : ⊥	pc <sub>u</sub> : 6 cnt <sub>u</sub> : ⊥	counter: 0
2	pc <sub>t</sub> : 7 cnt <sub>t</sub> : 0	pc <sub>u</sub> : 6 cnt <sub>u</sub> : ⊥	counter: 0
3	pc <sub>t</sub> : 8 cnt <sub>t</sub> : 0	pc <sub>u</sub> : 6 cnt <sub>u</sub> : ⊥	counter: 1
4	done	pc <sub>u</sub> : 6 cnt <sub>u</sub> : ⊥	counter: 1
5	done	pc <sub>u</sub> : 7 cnt <sub>u</sub> : 1	counter: 1
6	done	pc <sub>u</sub> : 8 cnt <sub>u</sub> : 1	counter: 2
7	done	done	counter: 2

# Trace abstractions



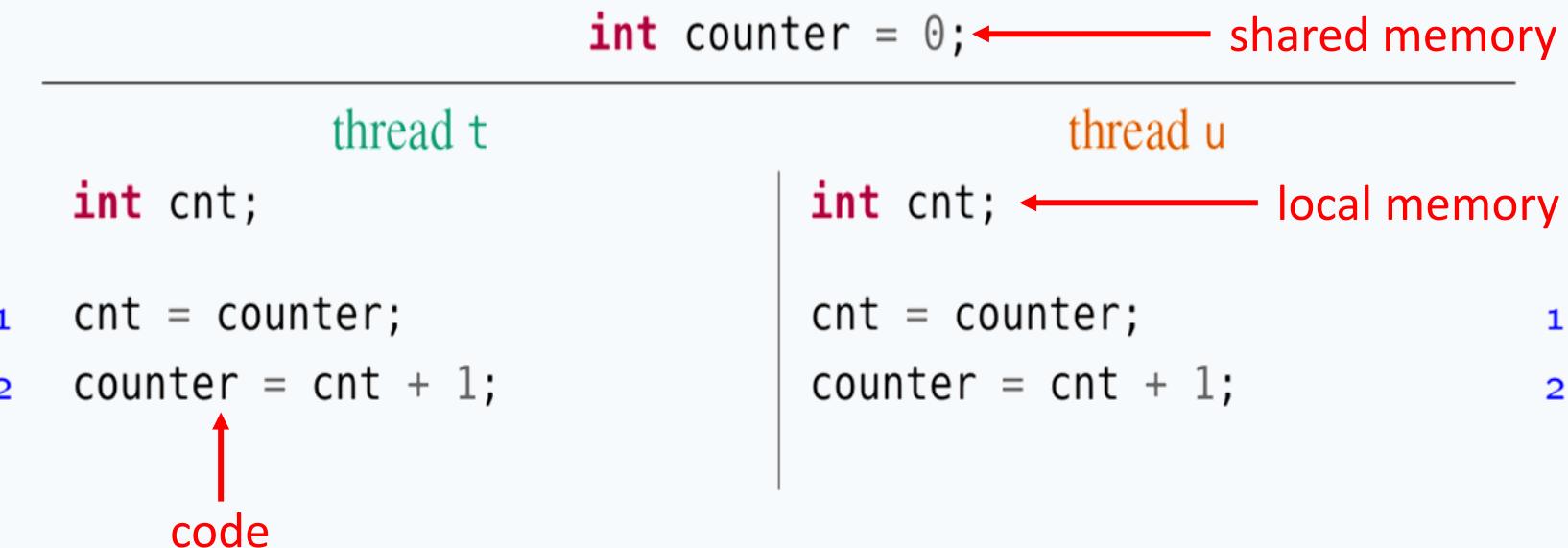
**Atomic/linearized:** The effects of each thread appear as if they happened **instantaneously**, when the trace snapshot is taken, in the thread's **sequential order**

**Complete:** The trace includes **all** intermediate **atomic states**

**Interleaved:** The trace is an **interleaving** of each thread's linear trace (in particular, no simultaneity)

# Abstraction of concurrent programs

When convenient, we will use an **abstract notation** for multi-threaded applications, which is similar to the pseudo-code used in Ben-Ari's book but uses Java syntax



Each line of code includes exactly one instruction that can be executed **atomically**:

- atomic statement  $\cong$  single read or write to global variable
- precise definition is tricky in Java, but we will learn to avoid pitfalls

© 2016–2019 Carlo A. Furia, Sandro Stucki



Except where otherwise noted, this work is licensed under the  
Creative Commons Attribution-ShareAlike 4.0 International License.

To view a copy of this license, visit

<http://creativecommons.org/licenses/by-sa/4.0/>.

# Races, locks and semaphores

TDA384/DIT391

Principles of Concurrent Programming

Nir Piterman and Gerardo Schneider

Chalmers University of Technology | University of Gothenburg



UNIVERSITY OF  
GOTHENBURG



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY

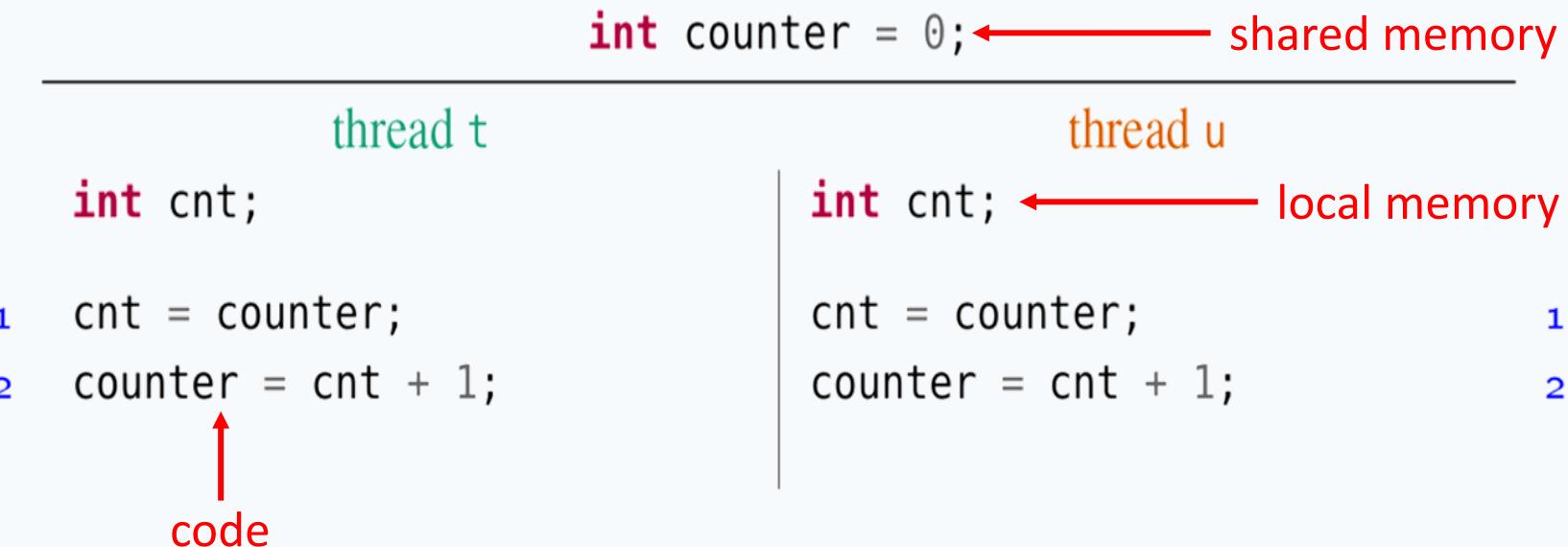
# Lesson's menu

- Concurrent programs and ConcurrentCounter (recap)
- What can be done?
  - Locks
  - Semaphores
- Theory and abstract problems
  - Races
  - Synchronization problems
- Synchronization with semaphores

# Concurrent programs

# Abstraction of concurrent programs

When convenient, we will use an **abstract notation** for multi-threaded applications, which is similar to the pseudo-code used in Ben-Ari's book but uses Java syntax.



Each line of code includes exactly one instruction that can be executed **atomically**:

- atomic statement  $\cong$  single read or write to global variable
- precise definition is tricky in Java, but we will learn to avoid pitfalls

# Traces

A sequence of **states** gives an execution **trace** of the concurrent program  
 (The program counter points to the atomic instruction that will be executed next)

```
int counter = 0;
```

	thread t	thread u	
#	t's LOCAL	u's LOCAL	SHARED
1	pc <sub>t</sub> : 6 cnt <sub>t</sub> : ⊥	pc <sub>u</sub> : 6 cnt <sub>u</sub> : ⊥	counter: 0
2	pc <sub>t</sub> : 7 cnt <sub>t</sub> : 0	pc <sub>u</sub> : 6 cnt <sub>u</sub> : ⊥	counter: 0
3	pc <sub>t</sub> : 7 cnt <sub>t</sub> : 0	pc <sub>u</sub> : 7 cnt <sub>u</sub> : 0	counter: 0
4	pc <sub>t</sub> : 7 cnt <sub>t</sub> : 0	pc <sub>u</sub> : 8 cnt <sub>u</sub> : 0	counter: 1
5	pc <sub>t</sub> : 8 cnt <sub>t</sub> : 0	done	counter: 1
6	done	done	counter: 1

One trace  
 (One possible  
 Interleaving)

# Concurrent counter

```
public class CCounter
    extends Counter
    implements Runnable
{
    // threads
    // will execute
    // run()
}
```

```
public class ConcurrentCount {
    public static void main(String[] args) {
        CCounter counter = new CCounter();
        // threads t and u, sharing counter
        Thread t = new Thread(counter);
        Thread u = new Thread(counter);
        t.start(); // increment once
        u.start(); // increment twice
        try { // wait for t and u to terminate
            t.join(); u.join();
        } catch (InterruptedException e) {
            System.out.println("Interrupted!");
        } // print final value of counter
        System.out.println(counter.counter());
    }
}
```

Prints different values in different runs!

# Is all lost?

- Introducing:

- Locks
- Semaphores

“magical” shared memory objects that achieve the impossible.

- For some internal details see Lecture 03 ...

# Locks

# Lock objects

A **lock** is a data structure with interface:

```
interface Lock {  
    void lock();           // acquire lock  
    void unlock();         // release lock  
}
```

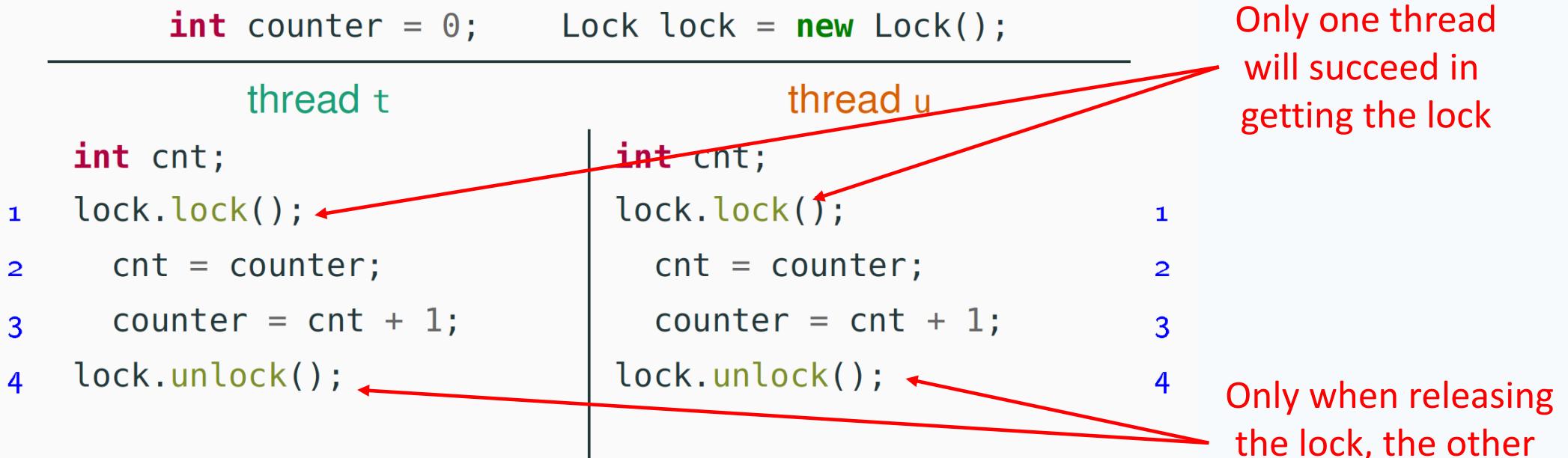
- Several threads **share** the same object lock of type Lock
- Many threads calling `lock.lock()`: exactly one thread  $t$  **acquires** the lock
  - $t$ 's call `lock.lock()` returns:  $t$  is holding the lock
  - other threads **block** on the call `lock.lock()`, waiting for the lock to become available
- A thread  $t$  that is holding the lock calls `lock.unlock()` to **release** the lock
  - $t$ 's call `lock.unlock()` returns: the lock becomes available
  - another thread waiting for the lock may succeed in acquiring it

Locks are also called **mutexes** (they guarantee mutual exclusion)

# Using locks

With lock objects ensuring no interference is trivial:

- Before: call `lock.lock()`
- After: call `lock.unlock()`



The implementation of the `Lock` interface should **guarantees mutual exclusion** and more (**deadlock freedom & starvation freedom**)

# Using locks in Java

```
// package with lock-related classes
import java.util.concurrent.locks.*;

// shared with other synchronizing threads
Lock lock;

lock.lock();           // entry protocol
try {
    // code that needs to be run in
    // mutual exclusion. Guaranteed
    // by the lock protocol
}

finally { // lock released even if an exception
    // is thrown above
    lock.unlock(); // exit protocol
}
```

Why is this  
inside a try-finally?

To avoid holding the lock in  
case of an exception  
(blocking all other threads)



# Counter with mutual exclusion

```

public class LockedCounter extends CCounter
{
  @Override
  public void run() {
    lock.lock(); Entry protocol
    try {
      // int cnt = counter;
      // counter = counter + 1;
      super.run();
    }
    finally {
      lock.unlock(); Exit protocol
    }
  }
  // shared by all threads working
  // on this object
  private Lock lock = new ReentrantLock();
}

```

Run exclusively [ ]

The main is as before, but instantiates an object of class LockedCounter

- What is printed by running:  
java ConcurrentCount?
- May the printed value change  
in different reruns?

**NO: Always 2**

To allow threads lock a resource  
more than once

# Built-in locks in Java

Every object in Java has an implicit lock, which can be accessed using the keyword **synchronized**

**Method locking (synchronized methods):**

```
synchronized T m() {  
    // the exclusive code  
    // is the whole method body  
}
```

**Block locking (synchronized block):**

```
synchronized(this) {  
    // the exclusive code  
    // is the block's content  
}
```

**Every call to m implicitly:**

1. acquires the lock
2. executes m
3. releases the lock

**Every execution of the block implicitly:**

1. acquires the lock
2. executes the block
3. releases the lock

# Counter with mutual exclusion: with **synchronized**

```
public class SyncCounter
    extends CCounter
{
    @Override
    public synchronized
    void run() {
        // int cnt = counter;
        // counter = counter + 1;
        super.run();
    }
}
```

```
public class SyncBlockCounter
    extends CCounter
{
    @Override
    public void run() {
        synchronized (this) {
            // int cnt = counter;
            // counter = counter + 1;
            super.run();
        }
    }
}
```

# Lock implementations in Java

- Many implementations of locks in `java.util.concurrent.locks`.
- The most common implementation of the Lock interface in Java is `class ReentrantLock`.
- The lock used by `synchronized` methods and blocks have the **same behavior** as the `explicit locks`.
- Built-in locks, and all lock implementations in `java.util.concurrent.locks` are *re-entrant*: a thread holding a lock can lock it again without causing a deadlock!

# Semaphores



\* Photo: British railway semaphores David Ingham, 2008

# Semaphores

A (general/counting) **semaphore** is a data structure with interface:

```
interface Semaphore {  
    int count();      // current value of counter  
    void up();        // increment counter  
    void down();     // decrement counter  
}
```

Several threads share the same object `sem` of type `Semaphore`:

- initially `count` is set to a nonnegative value  $C$  (the **capacity**)
- a call to `sem.up()` *uninterruptedly* increments `count` by one
- a call to `sem.down()`: **waits** until `count` is positive, and then *uninterruptedly* decrements `count` by one

# Semaphores for permissions

A semaphore is often used to **regulate access permits** to a **finite** number of resources:

- the **capacity**  $C$  is the number of initially available resources
- up (also called signal) **releases** a resource, which becomes available
- down (also called wait) **acquires** a resource if it is available

Example: **hot desks**

# Counter with mutual exclusion: with **semaphores**

Semaphores can be used to ensure no interference:

- initialize semaphore to 1
- **Before**: call sem.down()
- **After**: call sem.up()

Semaphore sem = <b>new Semaphore(1);</b>	
thread t	thread u
int cnt;  1 sem.down(); 2 cnt = counter; 3 counter = cnt + 1; 4 sem.up();	int cnt;  1 sem.down(); 2 cnt = counter; 3 counter = cnt + 1; 4 sem.up();

Acts as a lock

# Invariants

An object's **invariant** is a property that always holds between calls to the object's methods:

- the invariant holds *initially* (when the object is created)
- every method call *starts* in a state that satisfies the invariant
- every method call *ends* in a state that satisfies the invariant

Ex: A **bank account** that cannot be overdrawn has an **invariant**  $\text{balance} \geq 0$

```
class BankAccount {  
    private int balance = 0;  
    void deposit(int amount)  
    { if (amount > 0) balance += amount; }  
    void withdraw(int amount)  
    { if (amount > 0 && balance > amount) balance -= amount; }  
}
```

# Invariants in pseudo-code

- We may annotate classes with the pseudo-code keyword **invariant**
- Note that **invariant** is **not** a valid Java keyword – we highlight it in a different color – but we will use it whenever it helps make more explicit the behavior of classes

```
class BankAccount {  
    private int balance = 0;  
    void deposit(int amount)  
    { if (amount > 0) balance += amount; }  
    void withdraw(int amount)  
    { if (amount > 0 && balance > amount) balance -= amount; }  
    invariant{ balance >= 0; } // not valid Java code  
}
```

# Invariants of semaphores

A **semaphore** object with *initial capacity* C satisfies the invariant:

```
interface Semaphore {  
    int count();  
    void up();  
    void down();  
  
    invariant{  
        count () >= 0;  
        count () == C + #up - #down;  
    }  
}
```

Number of calls to up

up can increment  
beyond the initial capacity

Number of calls to down

NOT  
valid  
Java code

Invariants **characterize** the behavior of an object, and are very useful for **proofs**

# Binary semaphores

A **semaphore with capacity 1** and such that `count()` is always at most 1 is called a **binary semaphore**

```
interface BinarySemaphore extends Semaphore {  
    invariant  
    { 0 <= count() <= 1;  
     count() == C + #up - #down; }  
}
```

Mutual exclusion uses a  
binary semaphore:

```
Semaphore sem = new Semaphore(1);  
// shared by all threads
```

---

thread t

```
sem.down();  
// critical section  
sem.up();
```

# Binary semaphores vs. locks

Binary semaphores are very similar to *locks* with one difference:

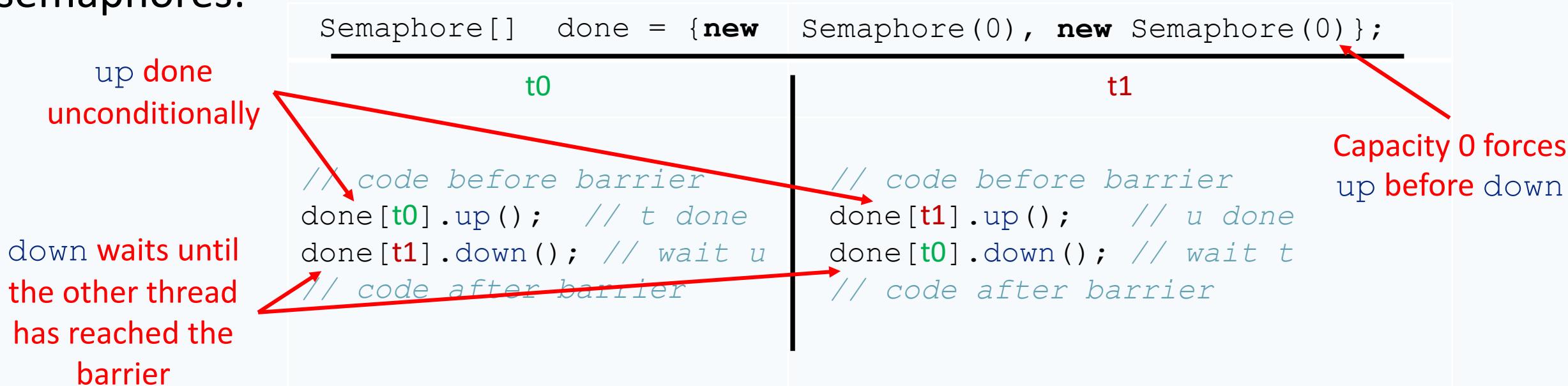
- In a *lock*, only the thread that decrements the counter to 0 can increment it back to 1
- In a *semaphore*, a thread may decrement the counter to 0 and then let another thread increment it to 1

Thus (binary) semaphores support transferring of permissions

# Is this re-usable?

A **barrier** is a form of synchronization where there is a *point* (the **barrier**) in a program's execution that all threads in a group have to reach **before any of them** is **allowed to continue**

A **solution** to the barrier synchronization problem for **2 threads** using binary semaphores:



down waits until  
the other thread  
has reached the  
barrier

# Using semaphores in Java

```
package java.util.concurrent;

public class Semaphore {
    Semaphore(int permits);
                    // initialize with capacity `permits'
    Semaphore(int permits, boolean fair);
                    // fair - explained later

    void acquire();           // corresponds to down
    void release();          // corresponds to up
    int availablePermits();  // corresponds to count
}
```

Method `acquire` may throw an `InterruptedException`: catch or propagate

# Races

# Race conditions

Concurrent programs are **nondeterministic**:

- Executing multiple times the same concurrent program with the same inputs may lead to **different execution traces**
- A result of the nondeterministic **interleaving** of each thread's trace to determine the overall program trace
- In turn, the interleaving is a result of the **scheduler**'s decisions

A **race condition** is a situation where the correctness of a concurrent program depends on the specific execution

The **concurrent counter** example has a **race condition**:

- in some executions the final value of counter is **2** (correct)
- in some executions the final value of counter is **1** (wrong)

Race conditions can greatly **complicate debugging!**

# Concurrency humor

A1: Knock Knock

A2: "Who's there?"

A1: "Race condition"

A1: Knock...

A2: "Who's there?"

A1: Knock...  
"Race condition"

A1: Knock Knock

A1: "Race condition"

A2: "Who's there?"

# Data races

Race conditions are typically caused by a **lack of synchronization** between threads that access **shared memory**

A **data race** occurs when two concurrent threads:

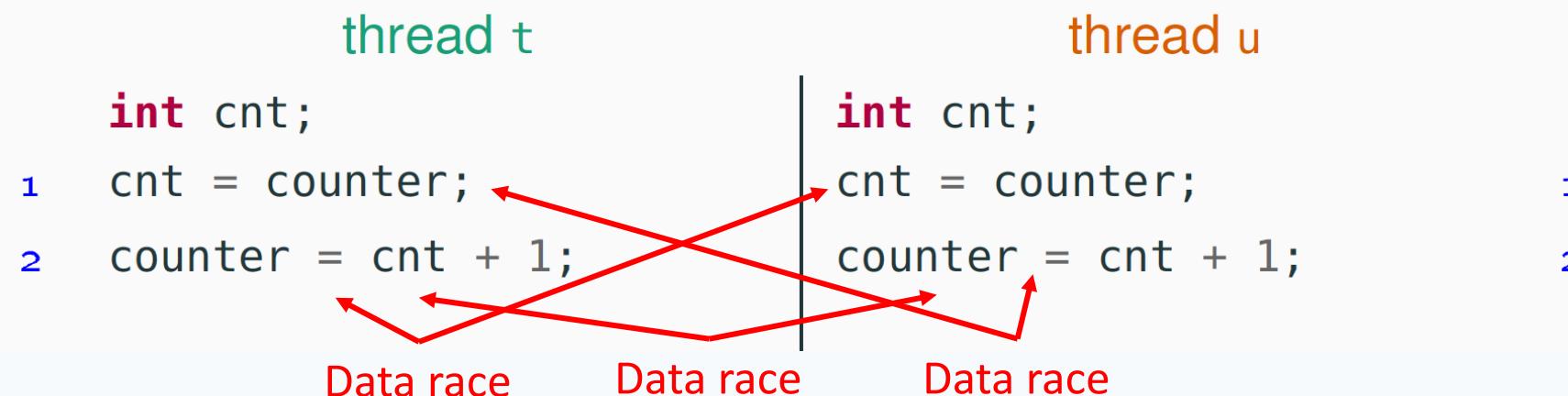
- Access a shared memory location
- At least one access is a **write**
- The threads use no explicit **synchronization mechanism** to protect the shared data

# Data races

A **data race** occurs when two concurrent threads:

- Access a shared memory location
  - At least one access is a **write**
  - The threads use no explicit **synchronization mechanism** to protect the shared data

```
int counter = 0;
```



# Data races vs. Race conditions

A **data race** occurs when two concurrent threads:

- Access a shared memory location
- At least one access is a **write**
- The threads use no explicit **synchronization mechanism** to protect the shared data

**Not every** race condition is a **data race**

**Not every** **data race** is a **race condition**

- Race conditions can occur even when there is no shared memory access
- Example: filesystems or network access

- The data race may not affect the result
- Example: if two threads write the same value to shared memory

# Abstract Synchronization problems

# Push out the races, bring in the speed

Concurrent programming introduces:

- the **potential** for parallel execution (faster, better resource usage)
- the **risk** of race conditions (incorrect, unpredictable computations)

The main challenge of concurrent programming is thus introducing parallelism without introducing race conditions

This requires to **restrict** the amount of nondeterminism by synchronizing processes/threads that access shared resources

# Synchronization

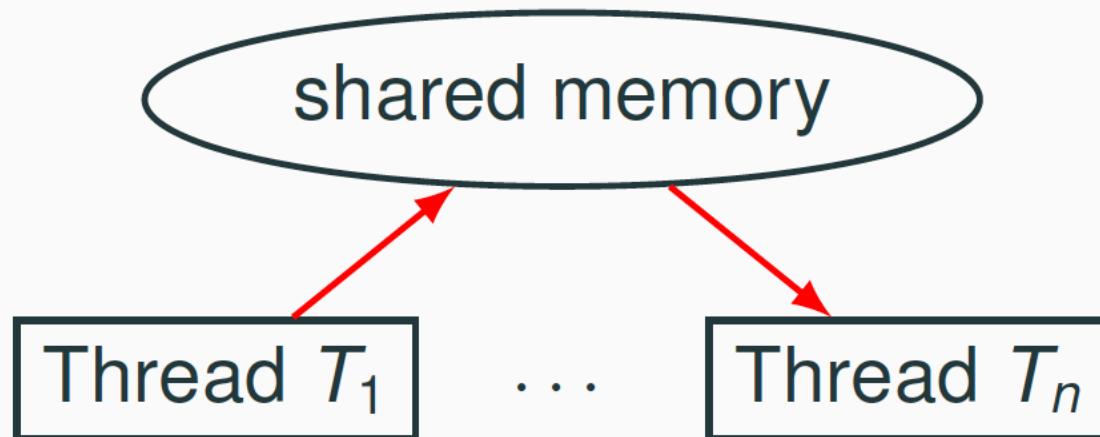
We will present several **synchronization** problems that often appear in concurrent programming, together with **solutions**

- **Correctness** (that is, avoiding race conditions) is **more important than performance**
  - An incorrect result that is computed faster is no good!
- However, we want to retain **as much concurrency as possible**
  - Otherwise we might as well stick with sequential programming

# Shared memory vs. Message passing synchronization

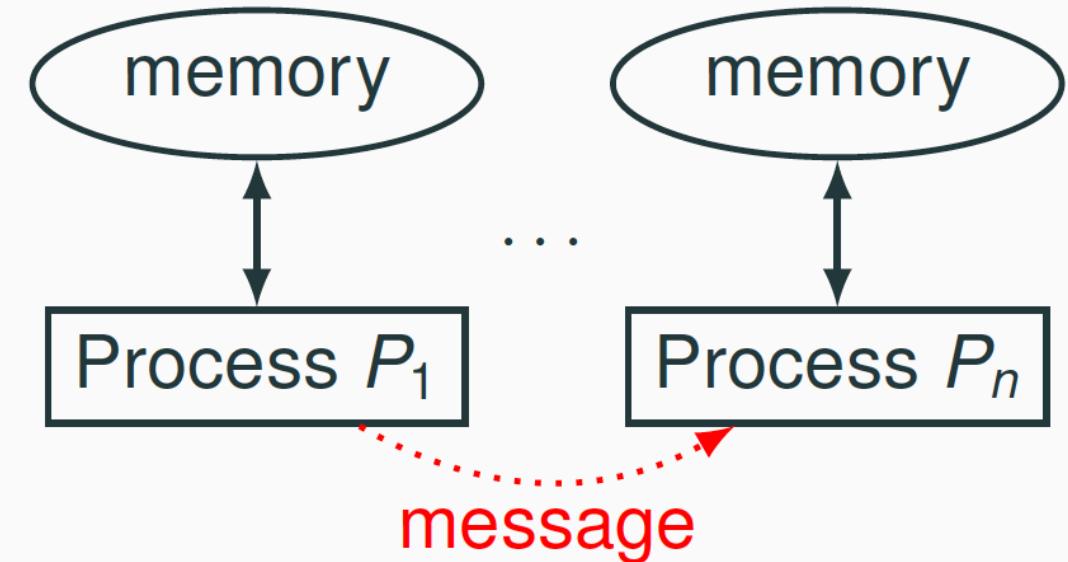
## Shared memory synchronization:

- Synchronize by **writing to and reading from shared memory**
- Natural choice in shared memory systems such as threads



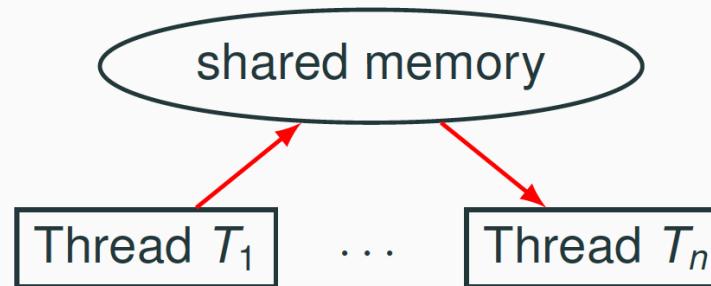
## Message passing synchronization:

- Synchronize by **exchanging messages**
- Natural choice in distributed memory systems such as processes

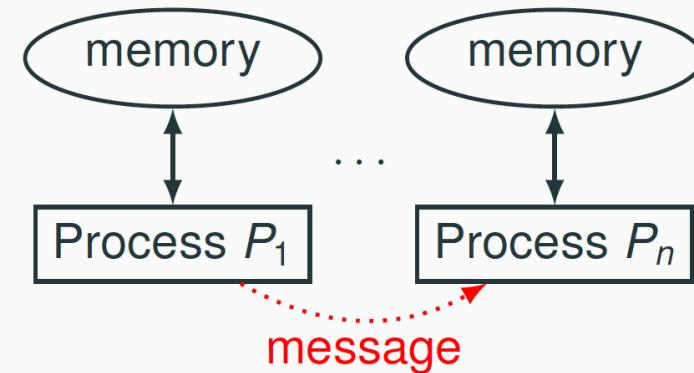


# Shared memory vs. Message passing synchronization

Shared memory synchronization:



Message passing synchronization:



The two synchronization models **overlap**:

- Send a message by writing to and reading from shared memory (ex: message board)
- Share information by sending a message (ex: order a billboard)
- We start by focusing on **shared memory concurrency**
- But the high-level abstraction applies to both

# The mutual exclusion problem

A fundamental synchronization problem which arises whenever multiple threads have access to a shared resource

**Critical Section:** Part of a program that accesses the shared resource (Ex: shared variable)

**Mutual Exclusion Property:** No more than 1 thread is in its critical section at any given time

**Mutual Exclusion Problem:** Devise a protocol for **accessing a shared resource** that satisfies the **mutual exclusion property**

Simplifications to present solutions in a uniform way:

- the critical section is an **arbitrary block** of code
- threads **continuously** try to enter the critical section
- threads spend a **finite amount of time** in the critical section
- we **ignore** what the threads do **outside** their critical sections

# The mutual exclusion problem

**Mutual Exclusion Problem:** Devise a protocol for accessing a shared resource that satisfies the **mutual exclusion property**

T <b>shared</b> ;	
thread $t_j$ <pre>// continuously <b>while</b> (<b>true</b>) {     entry protocol     critical section {         // access shared data     }     exit protocol } /* ignore behavior outside critical section */</pre>	thread $t_k$ <pre>// continuously <b>while</b> (<b>true</b>) {     entry protocol     critical section {         // access shared data     }     exit protocol } /* ignore behavior outside critical section */</pre>
<i>May depend on thread</i>	<i>Depends on computation</i>

# Mutual exclusion problem example: Concurrent Counter

Updating a **shared variable consistently** is an instance of the mutual exclusion problem

```
int counter = 0;
```

thread t

```
int cnt;  
  
while (true) {  
    entry protocol  
    critical section {  
        cnt = counter;  
        counter = cnt + 1;  
    }  
    exit protocol  
    return;  
}
```

thread u

```
int cnt;  
  
while (true) {  
    entry protocol  
    critical section {  
        cnt = counter;  
        counter = cnt + 1;  
    }  
    exit protocol  
    return;  
}
```

Take turns  
incrementing  
counter



# What's a good solution to the mutual exclusion problem?

A fully satisfactory solution is one that achieves **three properties**:

1. **Mutual exclusion**: at most one thread is in its critical section at any given time
2. **Freedom from deadlock**: if one or more threads try to enter the critical section, some thread will eventually succeed
3. **Freedom from starvation**: every thread that tries to enter the critical section will eventually succeed

A good solution should also work for an **arbitrary number of threads** sharing the same memory

(NOTE: **Freedom from starvation implies freedom from deadlock**)

# Deadlocks

A **deadlock** is the situation where a group of threads **wait forever** because each of them is waiting for resources that are held by another thread in the group (circular dependency)

- A mutual exclusion protocol provides **exclusive access** to shared resources to one thread at a time
- Threads that try to access the resource when it is not available will have to **block and wait**
- Mutually dependent waiting conditions may **introduce a deadlock**

# Deadlock: Example

A **deadlock** is the situation where a group of threads **wait forever** because each of them is waiting for resources that are held by another thread in the group (circular dependency)

A protocol that achieves mutual exclusion but introduces a deadlock:

**Entry protocol:** Wait until all other threads have executed their critical section



Via, resti servita Madama brillante – E. Tommasi Ferroni, 2012

# The Dining Philosophers

- **Dining philosophers**: A classic synchronization problem introduced by Dijkstra
- It illustrates the problem of deadlocks using a colorful metaphor (by Hoare)
- Five philosophers are sitting around a dinner table, with a fork in between each pair of adjacent philosophers
- Each philosopher alternates between thinking (**non-critical section**) and eating (**critical section**)
- In order to eat, a philosopher needs to pick up the two forks that lie to the philosopher's left and right
- Since the forks are **shared**, there is a **synchronization** problem between philosophers (**threads**)



# Deadlocking philosophers

An **unsuccessful attempt** at solving the dining philosophers problem:

```
entry () {  
    left_fork.acquire(); // pick up left fork  
    right_fork.acquire(); // pick up right fork  
}  
critical section { eat(); }  
exit () {  
    left_fork.release(); // release left fork  
    right_fork.release(); // release right fork  
}
```

This protocol **deadlocks** if all philosophers get their left forks, and wait forever for their right forks to become available



# The Coffman conditions

Necessary conditions for a **deadlock** to occur:

1. **Mutual exclusion**: threads may have exclusive access to the shared resources
  2. **Hold and wait**: a thread may request one resource while holding another one
  3. **No preemption**: resources cannot forcibly be released from threads that hold them
  4. **Circular wait**: two or more threads form a circular chain where each thread waits for a resource that the next thread in the chain is holding.
- \* Avoiding deadlocks requires to **break one or more** of these conditions

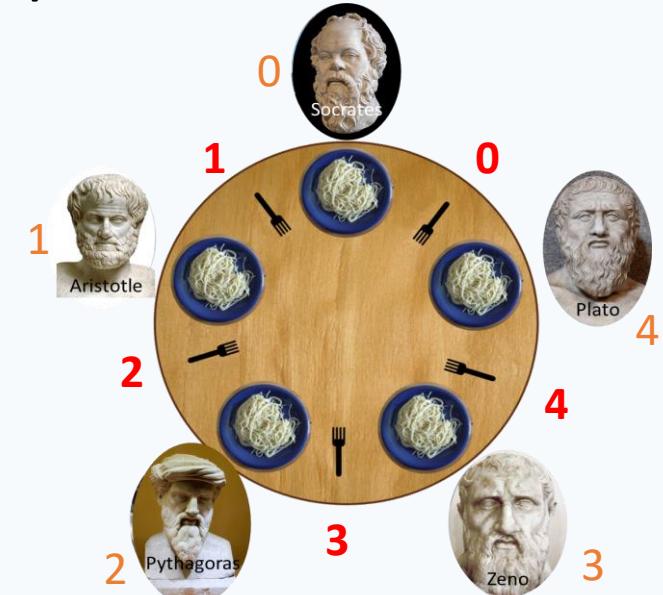
# Breaking a circular wait

A solution to the dining philosophers problem that **avoids deadlock** by **breaking circular wait**: pick up first the fork with the lowest *id* number

It avoids circular wait since not every philosopher will pick up their left fork first

```

entry () {
  if (left_fork.id() < right_fork.id())
    { left_fork.acquire();
      right_fork.acquire();
    }
  else
    { right_fork.acquire();
      left_fork.acquire();
    }
}
critical_section { eat(); }
exit () { /* ... */ }
  
```

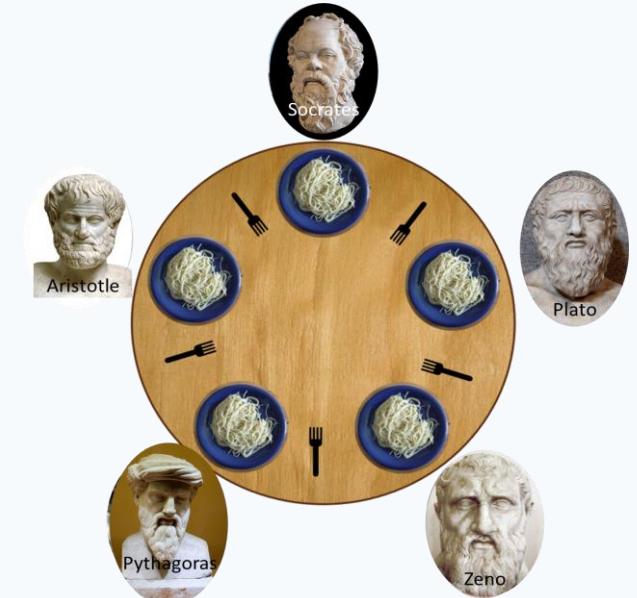


Ordering shared resources and forcing all threads to acquire the resources in order is a **common measure to avoid deadlocks**

# Starving philosophers

A solution to the dining philosophers problem that **avoids deadlock** by **breaking hold and wait** (and thus **circular wait**): pick up both forks at once (**atomic** op.)

```
entry () {  
    forks.acquire(); // pick up left and right  
                    // fork, atomically  
}  
critical section { eat(); }  
exit () {  
    forks.release(); // release left and right  
                    // fork, atomically  
}
```



It **avoids deadlock**, but it may **introduce starvation**: a philosopher may never get a chance to pick up the forks

# Starvation

**No deadlock** means that the system makes **progress as a whole**

However, some thread may still make no progress because it is **treated unfairly** in terms of access to shared resources

**Starvation** is the situation where a thread is  
**perpetually denied access** to a resource it requests

Avoiding starvation requires an additional assumption about the **scheduler**

# Fairness

**Starvation** is the situation where a thread is  
perpetually denied access to a resource it requests

Avoiding starvation requires the scheduler to  
**“give every thread a chance to execute”**

**Weak fairness:** if a thread continuously requests (that is, without interruptions) access to a resource, then access is granted eventually (or infinitely often)

**Strong fairness:** if a thread requests access to a resource infinitely often, then access is granted eventually (or infinitely often)

Applied to a scheduler:

- request = a thread is ready (**enabled**)
- fairness = every thread has a chance to execute

# Deadlock and Starvation in Java Locks

**class ReentrantLock**

**Mutual exclusion:**

- ReentrantLock guarantees mutual exclusion

**Starvation:**

- ReentrantLock does **not** guarantee freedom from starvation by default
- however, calling the constructor with new ReentrantLock(true) “favors granting access to the longest-waiting thread”
- this still does not guarantee that thread scheduling is fair

**Deadlocks:**

- one thread will succeed in acquiring the lock
- however, deadlocks may occur in systems that use multiple locks (remember the dining philosophers)

Explicit locks used by  
**synchronized** give no guarantee  
about starvation!

# Deadlock and Starvation in Semaphores

Every implementation of semaphores should **guarantee**:

- the **atomicity** of the up and down operations
- **deadlock freedom** (for one semaphore used correctly ...  
Deadlocks may still occur if there are other synchronization constraints!)

**Fairness** is optional:

**Weak semaphore**: threads waiting to perform down are scheduled **nondeterministically**

**Strong semaphore**: threads waiting to perform down are scheduled fairly in **FIFO** (First In First Out) order

# Mutex using binary semaphores

```
Semaphore sem = new Semaphore(1);  
// shared by all threads
```

thread t

```
sem.down();  
// critical section  
sem.up();
```

If the semaphore is **strong** this guarantees **starvation freedom**

# The $k$ -exclusion problem

The  **$k$ -exclusion** problem: devise a protocol that allows **up to  $k$  threads** to be in their **critical sections at the same time**

- Mutual exclusion problem = 1-exclusion problem
- The “hot desk” is an instance of the  $k$ -exclusion problem

A **solution** to the  $k$ -exclusion problem using a semaphore of capacity  $k$ : A straightforward generalization of mutual exclusion

```
Semaphore sem = new Semaphore(k)  
// shared by all threads
```

---

thread t

```
sem.down();  
// critical section  
sem.up();
```

# Models of concurrency & synchronization algorithms

**Lesson 3** of TDA384/DIT391

Principles of Concurrent Programming

Nir Piterman and Gerardo Schneider

Chalmers University of Technology | University of Gothenburg



UNIVERSITY OF  
GOTHENBURG



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY

# Lesson's menu

- Analyzing concurrency
- Mutual exclusion with only atomic reads and writes
  - Three **failed** attempts
  - Peterson's algorithm
  - Mutual exclusion with bounded waiting
- Implementing mutual exclusion algorithms in Java
- Implementing semaphores

# Lesson's menu

- Analyzing concurrency
  - Evaluate correctness of solutions
  - Important for understanding of race conditions
- Mutual exclusion with only atomic reads and writes
  - Understand the issues and problems
    - Interleaving and races
    - Why stronger synchronization
  - What's not working and what's working
- Implementing mutual exclusion algorithms in Java
  - Better understanding of Java memory model
  - More language constructs
  - Understanding exact behavior
- Implementing semaphores
  - Understand exact behavior
  - Demonstrate issues and problems

*Knowledge and understanding:*

- demonstrate knowledge of the issues and problems that arise in writing correct concurrent programs;
- identify the problems of synchronization typical of concurrent programs, such as race conditions and mutual exclusion

*Skills and abilities:*

- apply common patterns, such as lock, semaphores, and message-passing synchronization for solving concurrent program problems;
- apply practical knowledge of the programming constructs and techniques offered by modern concurrent programming languages;
- implement solutions using common patterns in modern programming languages

*Judgment and approach:*

- evaluate the correctness, clarity, and efficiency of different solutions to concurrent programming problems;
- judge whether a program, a library, or a data structure is safe for usage in a concurrent setting;
- pick the right language constructs for solving synchronization and communication problems between computational units.

# Analyzing concurrency

# State/transition diagrams

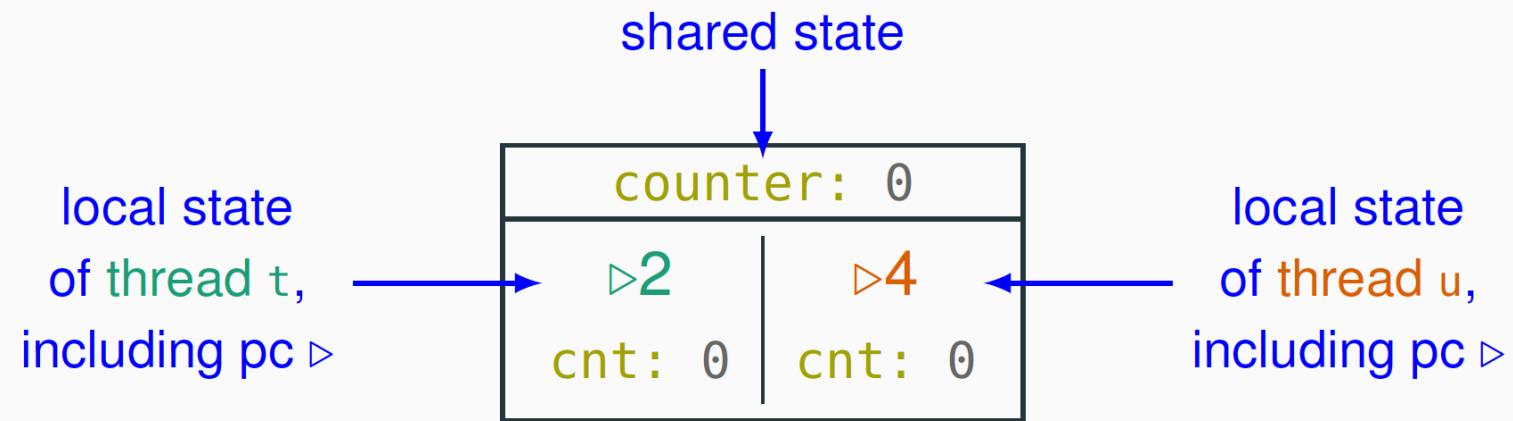
We capture essential elements of concurrent programs using **state/transition diagrams**

- Also called: *(finite) state automata*, *(finite) state machines*, or *transition systems*
- **States** in a diagram capture possible program states
- **Transitions** connect states according to execution order

**Structural properties** of a diagram capture semantic properties of the corresponding program

# States

A **state** captures the shared and local states of a concurrent program:



`int counter = 0;`

---

thread t

```
int cnt;  
  
1 cnt = counter;  
2 counter = cnt + 1;
```

thread u

```
int cnt;  
  
3 cnt = counter;  
4 counter = cnt + 1;
```

3

4

# States

A **state** captures the shared and local states of a concurrent program:

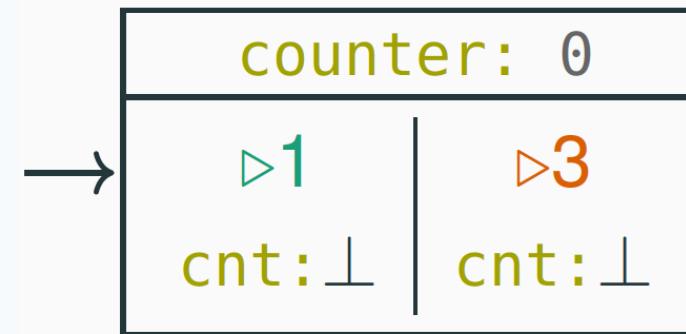
counter: 0	
▷2	▷4
cnt: 0	cnt: 0

When unambiguous, we simplify a state with only the **essential information**:

0	
▷2	▷4
0	0

# Initial states

The **initial state** of a computation is marked with an incoming arrow:



`int counter = 0;`

---

thread t

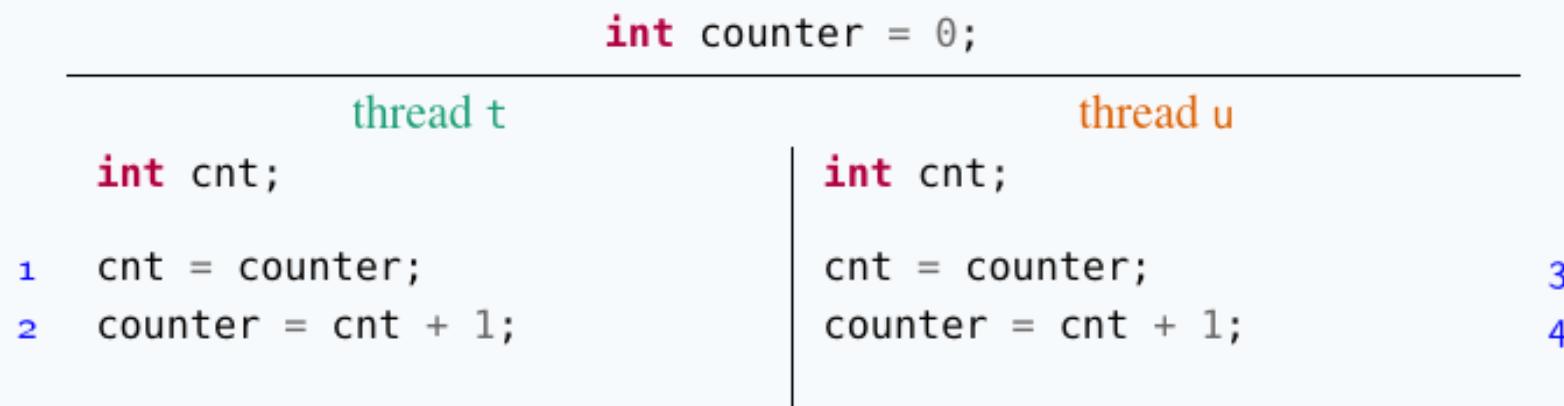
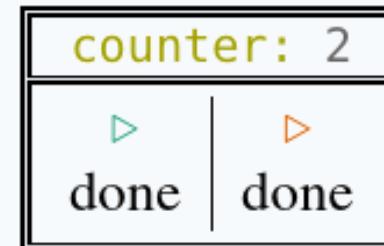
```
int cnt;  
  
1  cnt = counter;  
2  counter = cnt + 1;
```

thread u

```
int cnt;  
  
cnt = counter;          3  
counter = cnt + 1;     4
```

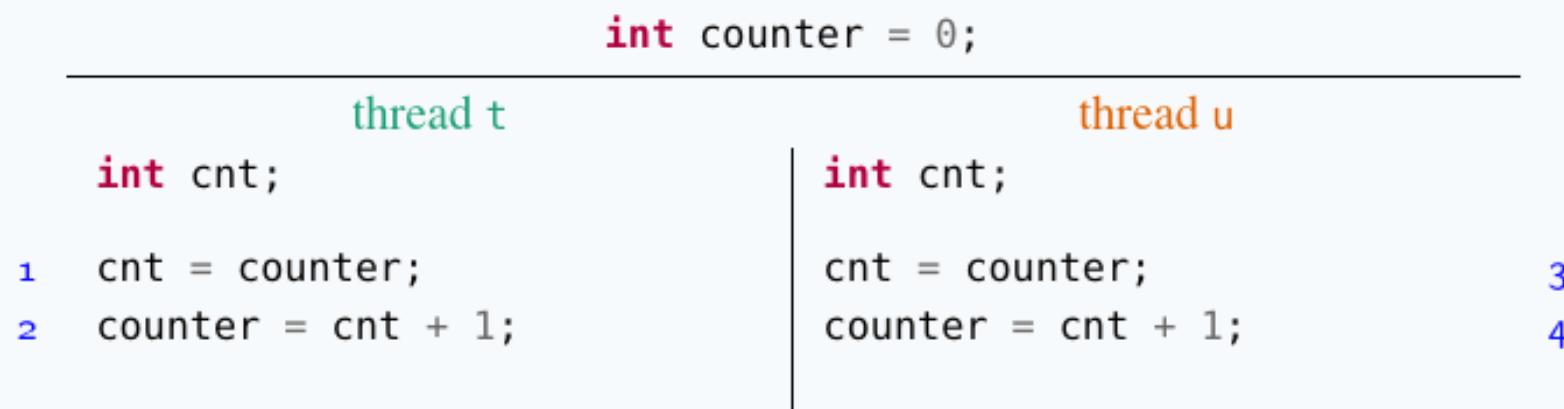
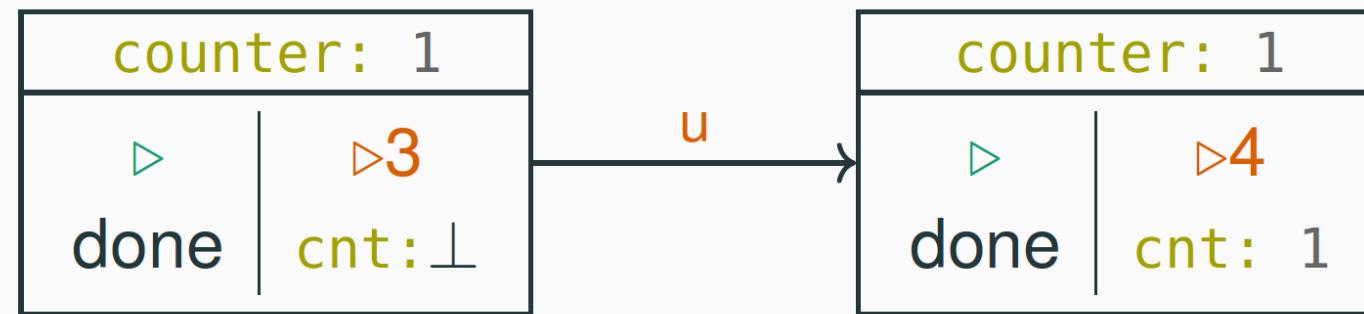
# Final states

The **final states** of a computation – where the program terminates – are marked with double-line edges:



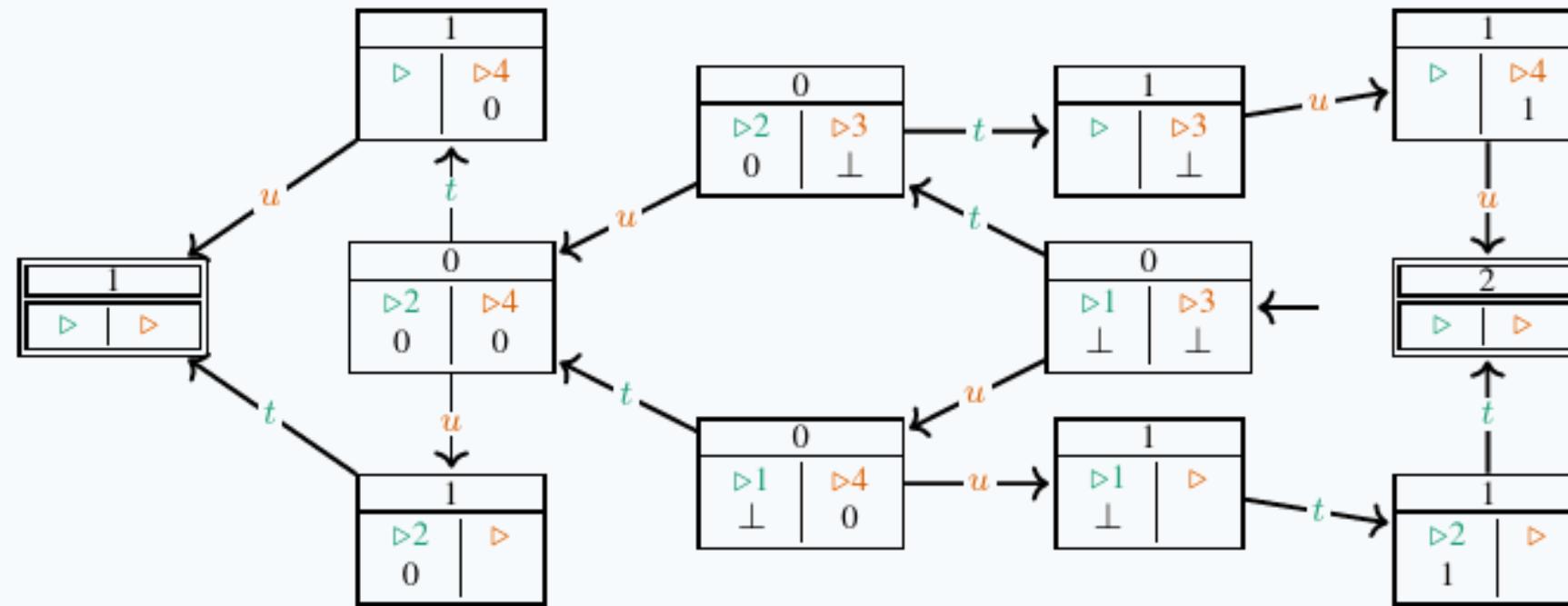
# Transitions

A **transition** corresponds to the execution of one atomic instruction, and it is an arrow connecting two states (or a state to itself):



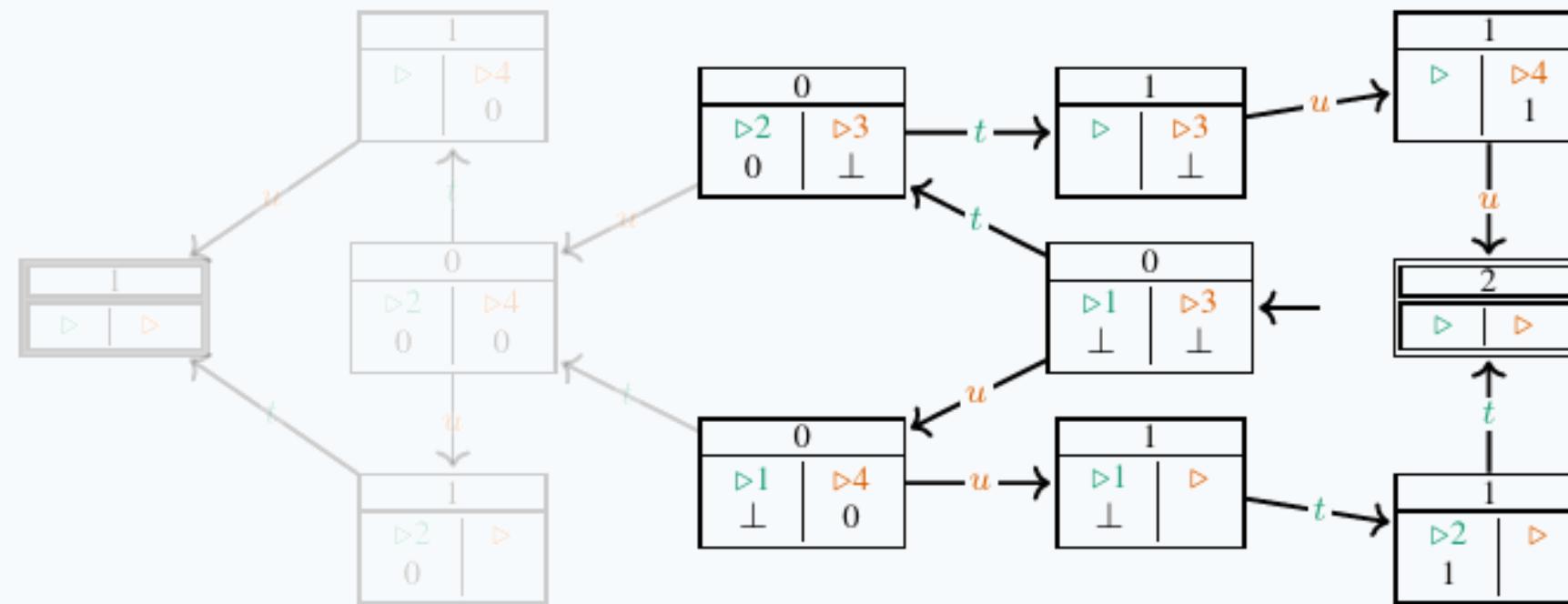
# A complete state/transition diagram

The **complete state/transition diagram** for the concurrent counter example explicitly shows **all possible interleavings**:



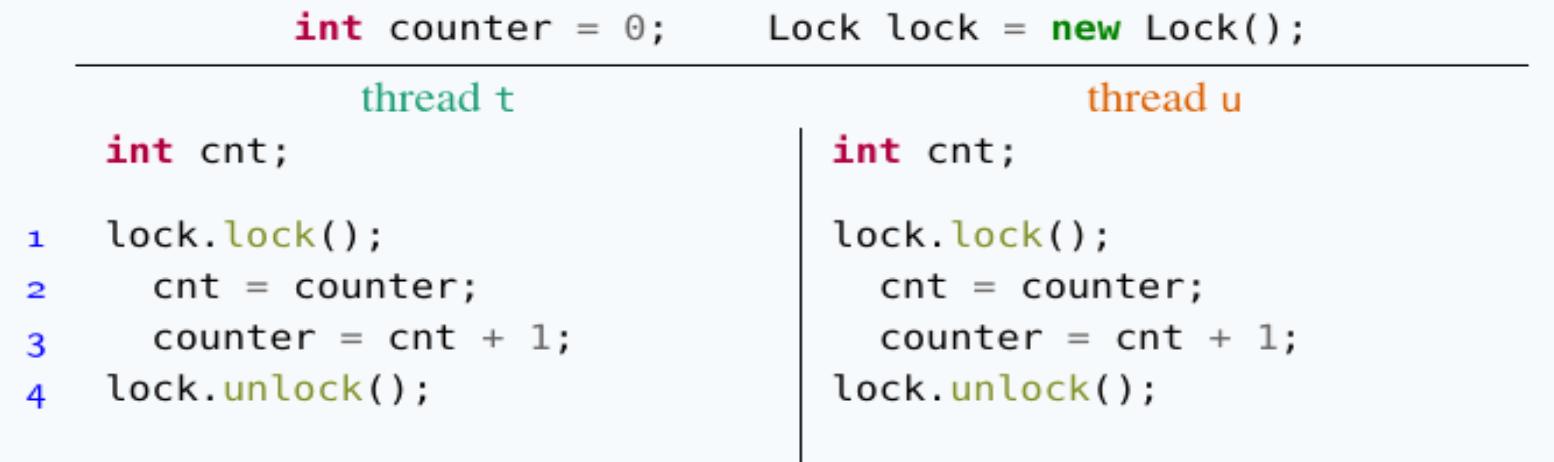
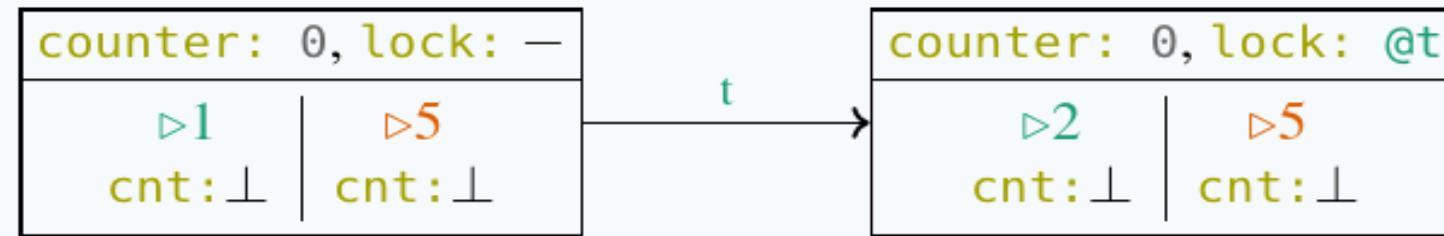
# State/transition diagram with locks?

The state/transition diagram of the concurrent counter example we would like to achieve using **locks**:



# Locking

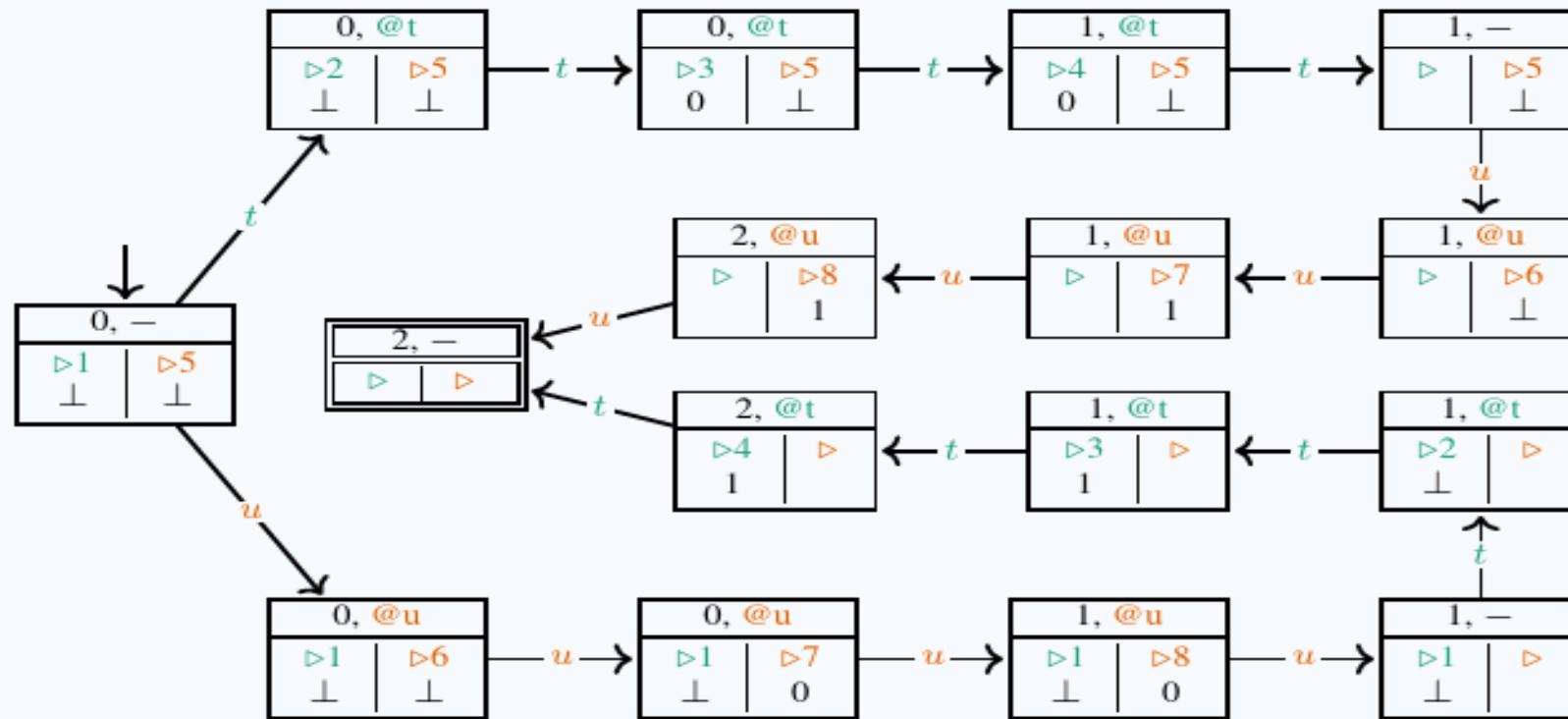
Locking and unlocking are considered atomic operations



This transition is only allowed if the lock is not held by another thread

# Counter with locks: state/transition diagram

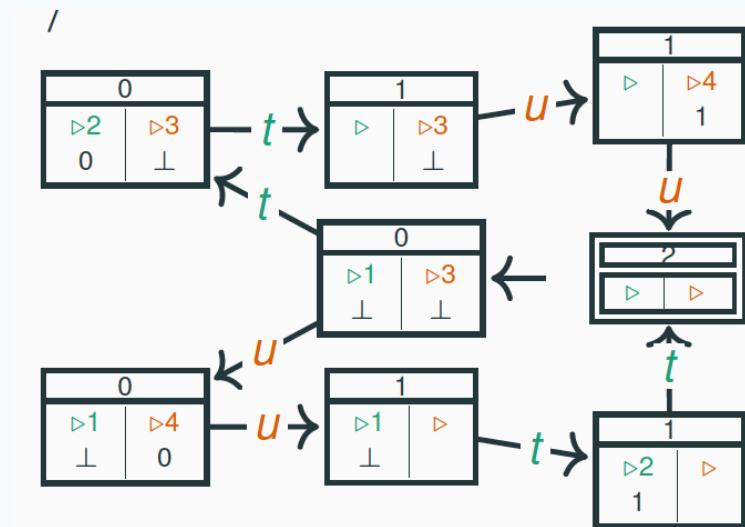
The state/transition diagram of the concurrent counter example **using locks** should contain **no (states representing) race conditions**:



# Transition tables

Transition tables are *equivalent representations* of the information of state/transition diagrams

CURRENT	NEXT WITH $t$	NEXT WITH $u$
$\langle 0, \triangleright 1, \perp, \triangleright 3, \perp \rangle$	$\langle 0, \triangleright 2, 0, \triangleright 3, \perp \rangle$	$\langle 0, \triangleright 1, \perp, \triangleright 4, 0 \rangle$
$\langle 0, \triangleright 2, 0, \triangleright 3, \perp \rangle$	$\langle 1, \triangleright , , \triangleright 3, \perp \rangle$	—
$\langle 0, \triangleright 1, \perp, \triangleright 4, 0 \rangle$	—	$\langle 1, \triangleright 1, \perp, \triangleright , \rangle$
$\langle 1, \triangleright , , \triangleright 3, \perp \rangle$	—	$\langle 1, \triangleright , , \triangleright 4, 1 \rangle$
$\langle 1, \triangleright 1, \perp, \triangleright , \rangle$	$\langle 1, \triangleright 2, 1, \triangleright , \rangle$	—
$\langle 1, \triangleright , , \triangleright 4, 1 \rangle$	—	$\langle 2, \triangleright , , \triangleright , \rangle$
$\langle 1, \triangleright 2, 1, \triangleright , \rangle$	$\langle 2, \triangleright , , \triangleright , \rangle$	—
$\langle 2, \triangleright , , \triangleright , \rangle$	—	—



# Reasoning about program properties

The **structural properties** of a diagram capture semantic properties of the program:

**Mutual exclusion:** there are no states where two threads are in their critical section

**Deadlock freedom:** for every (non-final) state, there is an outgoing transition

**Starvation freedom:** there is no (looping) path such that a thread never enters its critical section while trying to do so

**No race conditions:** all the final states have the same (correct) result

- We will build and analyze state/transition diagrams only for simple examples, since it quickly becomes tedious
- **Model checking** is a technique that automates the construction and analysis of state/transition diagrams with billions of states
  - We'll give a short introduction to model checking in one of the last classes

Mutual exclusion with only  
atomic reads and writes

# Locks: recap

A **lock** is a data structure (an **object** in Java) with interface:

```
interface Lock {  
    void lock();      // acquire lock  
    void unlock();   // release lock  
}
```

- Several threads share the same object **lock** of type **Lock**
- Threads calling **lock.lock()**: exactly one thread  $t$  **acquires** the lock:
  - $t$ 's call **lock.lock()** returns:  $t$  is **holding** the lock
  - other threads **block** on the call **lock.lock()**, waiting for the lock to become available
- A thread  $t$  that is holding the lock calls **lock.unlock()** to **release** the lock:
  - $t$ 's call **lock.unlock()** returns: the lock becomes **available**
  - another thread **waiting** for the lock may succeed in acquiring it

# Mutual exclusion without locks

Can we implement locks using **only** atomic instructions – reading and writing shared variables?

- It is possible
- But it is also tricky!



- We present some **classical algorithms** for mutual exclusion using only **atomic reads and writes**
  - The presentation builds up to the correct algorithms in a series of attempts, which highlight the principles that underlie how the algorithms work

# The mutual exclusion problem - recap

Given  $N$  threads, each executing:

```
// continuously
while (true) {
    entry protocol ←
    critical section {
        // access shared data
    }
    exit protocol ←
} /* ignore behavior
outside critical section */
```

Now protocols can use  
only reads and writes  
of shared variables

Design the entry and exit protocols to ensure:

- mutual exclusion
- freedom from deadlock
- freedom from starvation

Initially we limit ourselves to  $N = 2$  threads,  $t_0$  and  $t_1$

# Busy waiting

In the pseudo-code, we will use the shorthand

$$\text{await}(c) \triangleq \text{while } (!c) \{ \}$$

to denote **busy waiting** (also called **spinning**):

- keep reading shared variable `c` as long as it is `false`
- proceed when it becomes `true`
- Busy waiting is generally **inefficient** (unless typical waiting times are shorter than context switching times), so you should **avoid using it**
  - We use it only because it is a good device to illustrate the nuts and bolts of mutual exclusion protocols
- Note that `await` is **not** a valid Java keyword
  - We highlight it in a different color – but we will use it as a shorthand for better readability

Mutual exclusion with only  
atomic reads and writes

Three *failed* attempts

# Double-threaded mutual exclusion: First naive attempt

Use Boolean flags `enter[0]` and `enter[1]`:

- each thread `waits` until the other thread is `not trying` to enter the critical section
- before thread  $t_k$  is about `to enter` the critical section, it sets `enter[k]` to true

<code>boolean[] enter = {false, false};</code>	
thread $t_0$	thread $t_1$
<code>1 while (true) {</code>	<code>9 while (true) {</code>
<code>2   // entry protocol</code>	<code>10 // entry protocol</code>
<code>3   await (!enter[1]);</code>	<code>11 await (!enter[0]);</code>
<code>4   enter[0] = true;</code>	<code>12 enter[1] = true;</code>
<code>5   critical section { ... }</code>	<code>13 critical section { ... }</code>
<code>6   // exit protocol</code>	<code>14 // exit protocol</code>
<code>7   enter[0] = false;</code>	<code>15 enter[1] = false;</code>
<code>8 }</code>	<code>16 }</code>

# The first naive attempt is incorrect!

The **first attempt** does **not guarantee mutual exclusion**:  $t_0$  and  $t_1$  can be in the critical section at the same time

Both threads here! How?

```

boolean[] enter = {false, false};

thread t0                                thread t1
-----|-----|
1  while (true) {                         9  while (true) {
2    // entry protocol                      // entry protocol
3    await (!enter[1]);                   10
4    enter[0] = true;                     11
5    critical section { ... }             12
6    // exit protocol                      critical section { ... }
7    enter[0] = false;                    13
8  }                                         // exit protocol
                                              14
                                              enter[1] = false;
                                              15
                                              }
                                              16
  
```

#	$t_0$	$t_1$	SHARED
1	pc <sub>0</sub> : <code>await (!enter[1])</code>	pc <sub>1</sub> : <code>await (!enter[0])</code>	enter: <code>false, false</code>
2	pc <sub>0</sub> : <code>enter[0] = true</code>	pc <sub>1</sub> : <code>await (!enter[0])</code>	enter: <code>false, false</code>
3	pc <sub>0</sub> : <code>enter[0] = true</code>	pc <sub>1</sub> : <code>enter[1] = true</code>	enter: <code>false, false</code>
4	pc <sub>0</sub> : critical section	pc <sub>1</sub> : <code>enter[1] = true</code>	enter: <code>true, false</code>
5	pc <sub>0</sub> : critical section	pc <sub>1</sub> : critical section	enter: <code>true, true</code>

The **problem** seems to be that `await` is executed **before** setting `enter`, so one thread may proceed ignoring that the other thread is also proceeding

# Double-threaded mutual exclusion: Second naive attempt

When thread  $t_k$  wants to enter the **critical section**:

- it **first** sets `enter[k]` to true
- then it **waits** until the other thread is **not trying** to enter the critical section

```
boolean[] enter = {false, false};

---



| thread $t_0$                     | thread $t_1$ |
|----------------------------------|--------------|
| 1 <b>while</b> ( <b>true</b> ) { | 9            |
| 2 <i>// entry protocol</i>       | 10           |
| 3     enter[0] = <b>true</b> ;   | 11           |
| 4 <b>await</b> (!enter[1]);      | 12           |
| 5     critical section { ... }   | 13           |
| 6 <i>// exit protocol</i>        | 14           |
| 7     enter[0] = <b>false</b> ;  | 15           |
| 8   }                            | 16           |


```

# The second naive attempt may deadlock!

## The second attempt:

- **guarantees mutual exclusion**:  $t_0$  is in the critical section iff `enter[1]` is false, iff  $t_1$  has not set `enter[1]` to true, iff  $t_1$  has not entered the critical section ( $t_1$  has not executed line yet)
- does **not guarantee freedom from deadlocks**

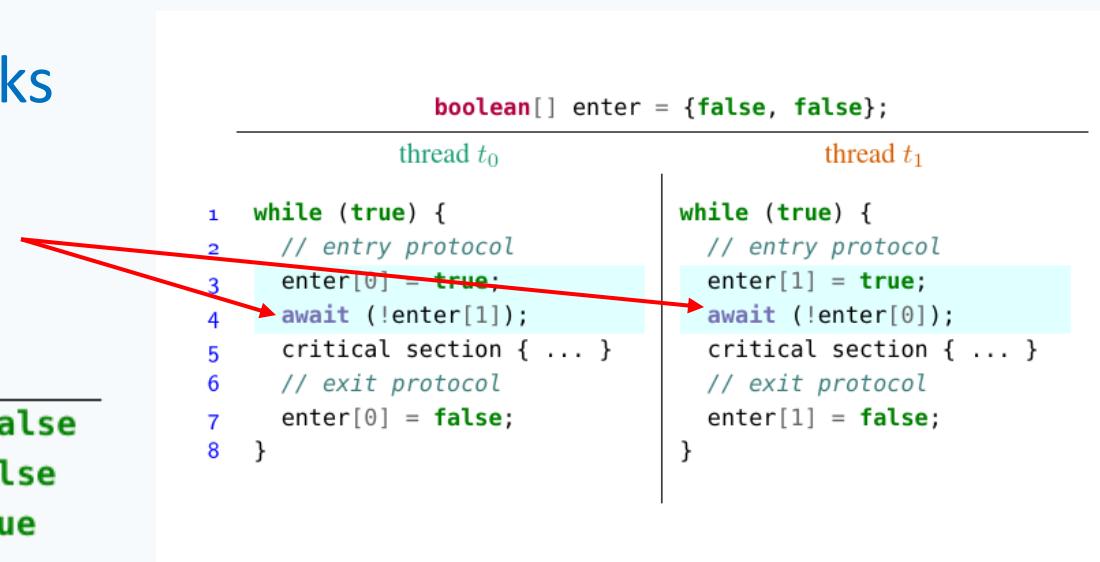
#	$t_0$	$t_1$	SHARED
1	<code>pc<sub>0</sub>: enter[0] = true</code>	<code>pc<sub>1</sub>: enter[0] = true</code>	<code>enter: false, false</code>
1	<code>pc<sub>0</sub>: await (!enter[1])</code>	<code>pc<sub>1</sub>: enter[0] = true</code>	<code>enter: true, false</code>
2	<code>pc<sub>0</sub>: await (!enter[1])</code>	<code>pc<sub>1</sub>: await (!enter[0])</code>	<code>enter: true, true</code>

Both threads might end up here, blocked. Why?

```

boolean[] enter = {false, false};

thread t0                                thread t1
-----|-----|-----|-----|-----|-----|
1  while (true) {                         9  while (true) {
2    // entry protocol                      // entry protocol
3    enter[0] = true;                     10  enter[1] = true;
4    await (!enter[1]);                  11  await (!enter[0]);
5    critical section { ... }           12  critical section { ... }
6    // exit protocol                      // exit protocol
7    enter[0] = false;                   13  enter[1] = false;
8  }                                         14
                                         15
                                         16
  
```



The **problem** seems to be that the **two variables** `enter[0]` and `enter[1]` are accessed independently

- each thread may be waiting for permission to proceed from the other thread

# Double-threaded mutual exclusion: Third naive attempt

Use one single integer variable `yield`:

- thread  $t_k$  waits for its turn while `yield` is  $k$
- when it is done with its critical section, it yields control to the other thread by setting `yield = k`

```
int yield = 0 || 1; // initialize to either value
```

thread  $t_0$

```
1 while (true) {  
2     // entry protocol  
3     await (yield != 0);  
4     critical section { ... }  
5     // exit protocol  
6     yield = 0;  
7 }
```

thread  $t_1$

```
8 while (true) {  
9     // entry protocol  
10    await (yield != 1);  
11    critical section { ... }  
12    // exit protocol  
13    yield = 1;  
14 }
```

# The third naive attempt may starve some thread!

## The third attempt:

- **guarantees mutual exclusion:**
  - $t_0$  is in the critical section iff `yield` is 1
  - iff `yield` was initialized to 1 or  $t_1$  has set `yield` to 1
  - iff  $t_1$  is not in the critical section ( $t_0$  has not executed line 6 yet).

- **guarantees freedom from deadlocks:** each thread enables the other thread, so that a circular wait is impossible
- does **not guarantee freedom from starvation:** if one stops executing in its **non-critical** section, the other thread will starve (after one last access to its critical section)

Later in the course: we will discuss how model checking can help to verify whether such correctness properties hold in a concurrent program

<code>int yield = 0    1; // initialize to either value</code>	
	thread $t_0$
	thread $t_1$
	8
1 <code>while (true) {</code>	9
2 <code>// entry protocol</code>	10
3 <code>await (yield != 0);</code>	11
4 <code>critical section { ... }</code>	12
5 <code>// exit protocol</code>	13
6 <code>yield = 0;</code>	14
7    }	}

# The third naive attempt may starve some thread!

```
int yield = 0 || 1; // initialize to either value
```

thread $t_0$	thread $t_1$
1 <b>while (true) {</b>	8 <b>while (true) {</b>
2     // entry protocol	9     // entry protocol
3 <b>await (yield != 0);</b>	10 <b>await (yield != 1);</b>
4     critical section { ... }	11   critical section { ... }
5     // exit protocol	12   // exit protocol
6 <b>yield = 0;</b>	13 <b>yield = 1;</b>
7 <b>}</b>	14 <b>}</b>

... then thread  $t_0$  will starve

If  $\text{yield}=0$  and thread  $t_1$  stops executing here (before the entry protocol)...

# Peterson's algorithm

# Peterson's algorithm

Combine the ideas behind the second and third attempts:

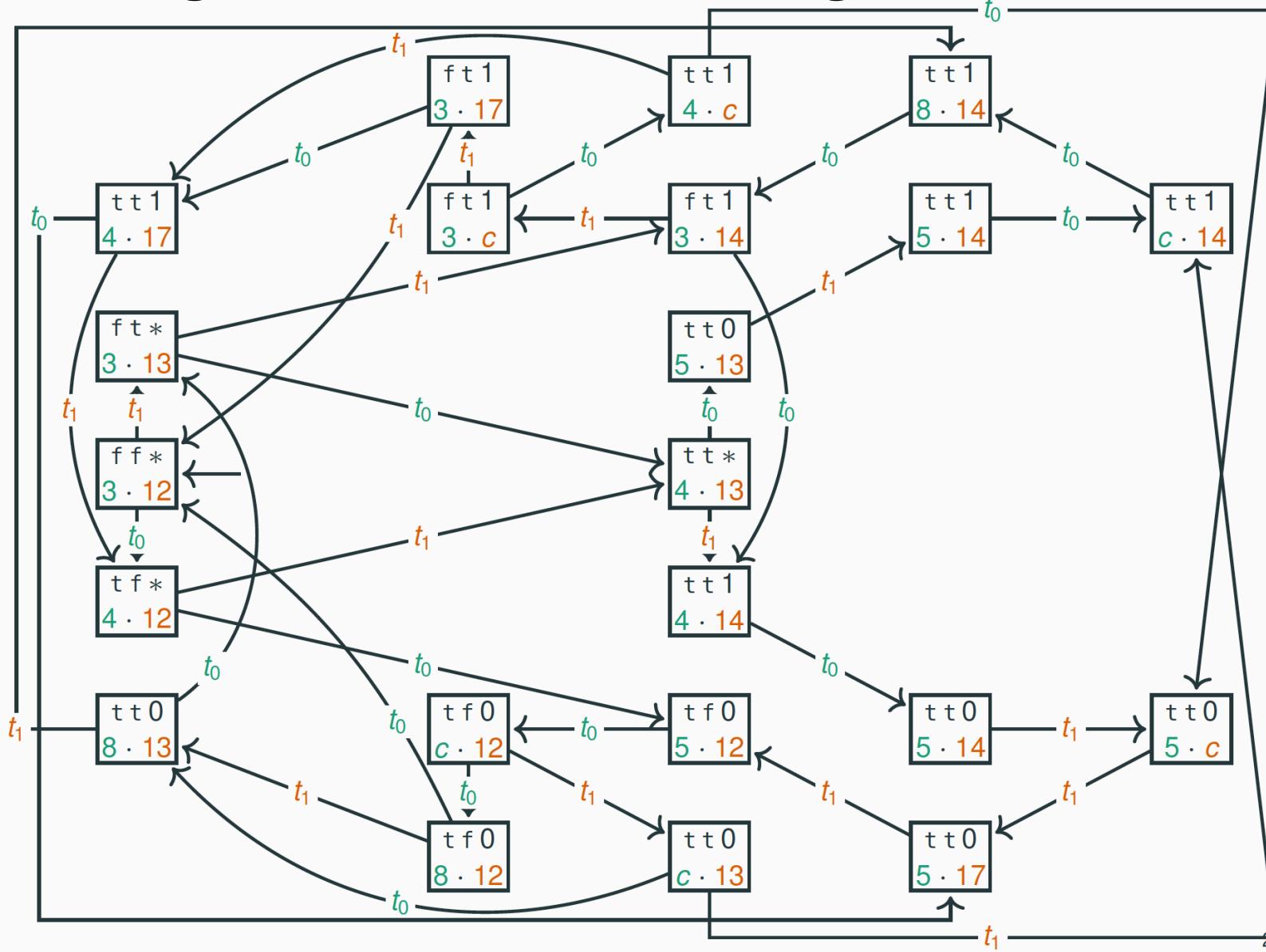
- thread  $t_k$  first sets `enter[k]` to true
- but lets the other thread go first – by setting `yield`

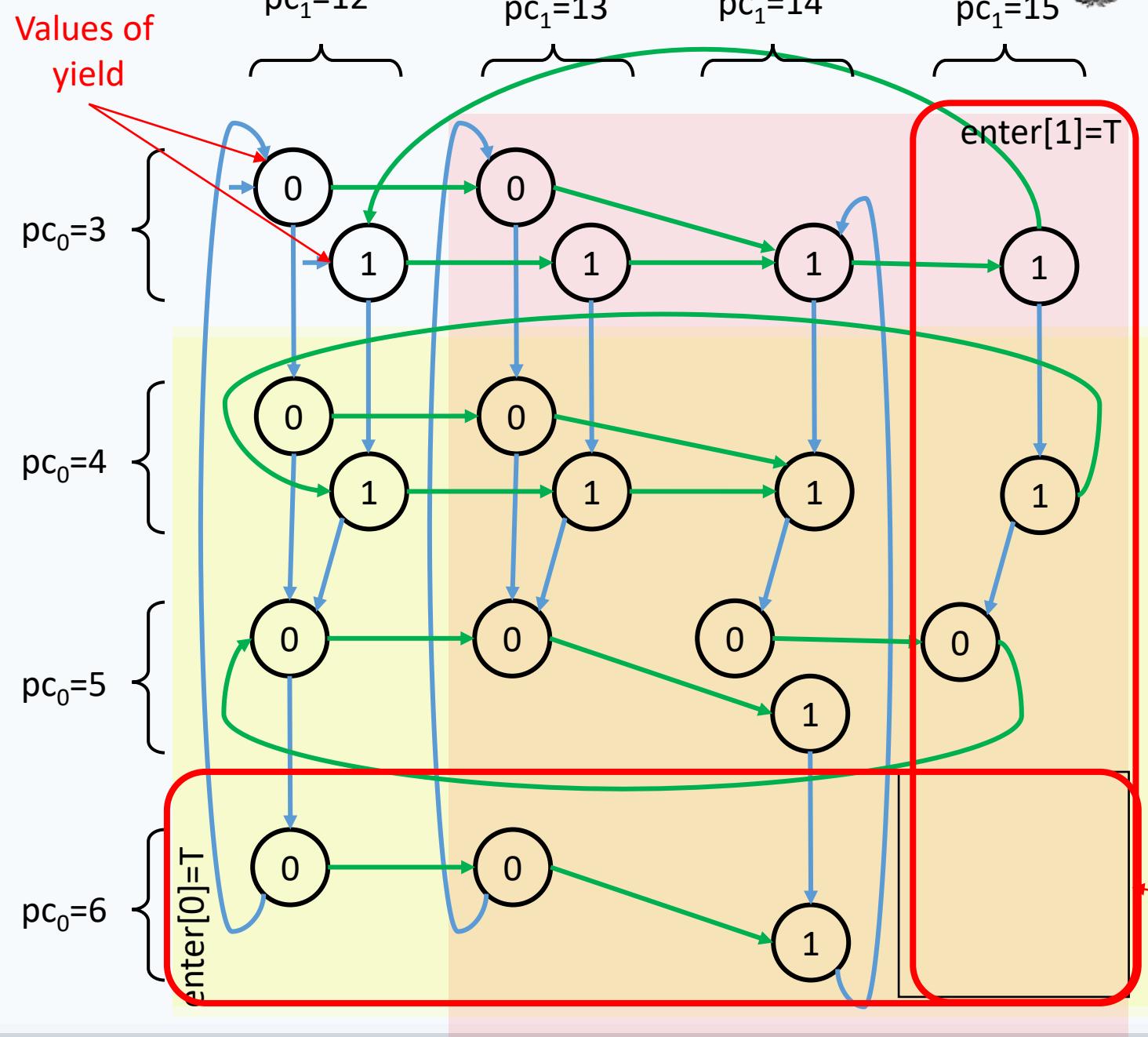
Equivalent to:  
`wait while  
 (enter[1]=true  
 &  
 yield=0)`

Enter only when  
`(enter[1]=false  
 OR  
 yield=1)`

		<code>boolean[] enter = {false, false}; int yield = 0    1;</code>
	thread $t_0$	thread $t_1$
	<pre> 1  while (true) { 2    // entry protocol 3    enter[0] = true; 4    yield = 0; 5    await (!enter[1]    yield != 0); 6    critical section { ... } 7    // exit protocol 8    enter[0] = false; 9  }</pre>	<pre> 10 11 12 13 14 15 16 17 18</pre>
		<p>Works even if two reads are non-atomic</p>

# State/transition diagram of Peterson's algorithm





# Another state/transition diagram of Peterson's algorithm

```

boolean[] enter = {false, false};    int yield = 0 || 1;

```

---

<b>thread <i>t</i><sub>0</sub></b>	<b>thread <i>t</i><sub>1</sub></b>
1 while (true) {	10
2 // entry protocol	11
3 enter[0] = true;	12
4 yield = 0;	13
5 await (!enter[1]	14
6 yield != 0);	
7 critical section { ... }	15
8 // exit protocol	16
9 enter[0] = false;	17
}	18

Omitting lines  
7-9 and 16-18

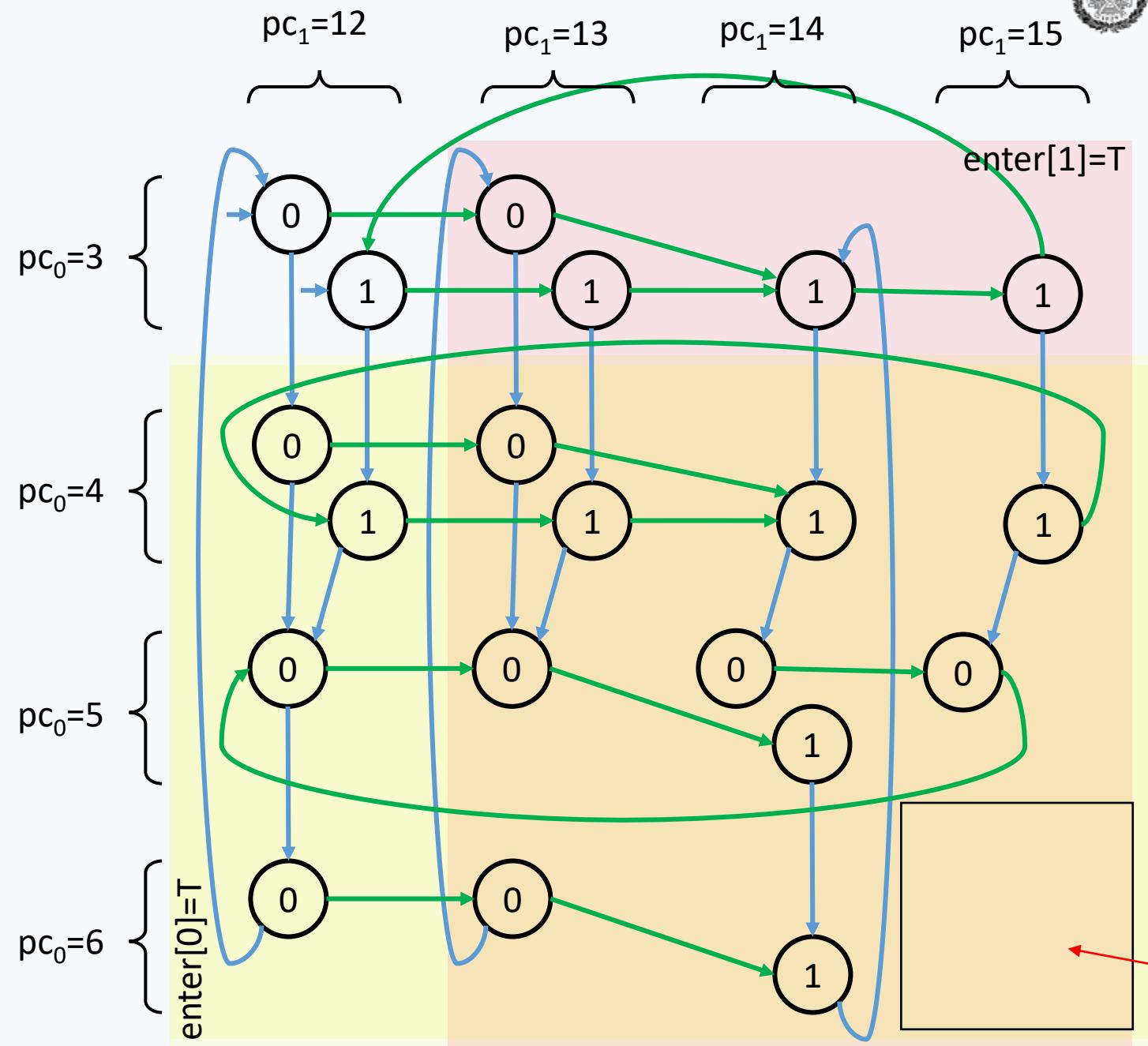
# Checking the correctness of Peterson's algorithm

By inspecting the state/transition diagram, we can check that Peterson's algorithm satisfies:

**mutual exclusion:** there are no states where both threads are at  $pc_0=6$  and  $pc_1=15$  (in the critical section)

**deadlock freedom:** every state has at least one outgoing transition

**starvation freedom:** if thread  $t_0$  is in its critical section, then thread  $t_1$  can reach its critical section without requiring thread  $t_0$ 's collaboration after  $t_0$  executes the exit protocol



Peterson's algorithm  
satisfies mutual exclusion  
and is deadlock free

---

<pre> boolean[] enter = {false, false};  int yield = 0    1; </pre>	<b>thread <math>t_0</math></b>	<b>thread <math>t_1</math></b>
---	--------------------------------	--------------------------------

```

1  while (true) {
2    // entry protocol
3    enter[0] = true;
4    yield = 0;
5    await (!enter[1]           ||
6           yield != 0);
7    critical section { ... }
8    // exit protocol
9    enter[0] = false;
}

```

Both in  
Critical Section

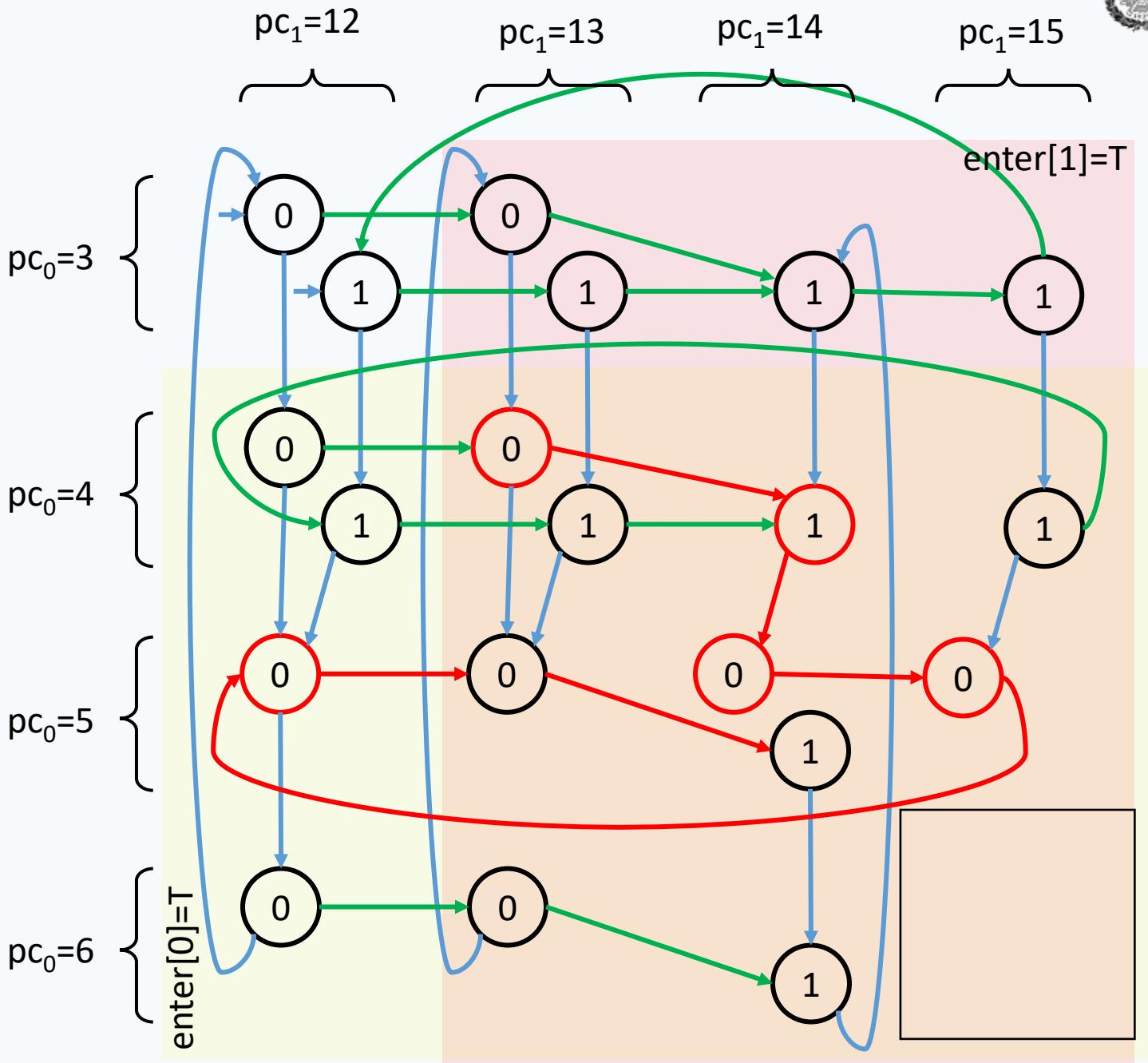
# Checking the correctness of Peterson's algorithm

By inspecting the state/transition diagram, we can check that Peterson's algorithm satisfies:

**mutual exclusion:** there are no states where both threads are at  $pc0=6$  and  $pc1=15$  (in the critical section)

**deadlock freedom:** every state has at least one outgoing transition

**starvation freedom:** if thread  $t_0$  is in its critical section, then thread  $t_1$  can reach its critical section without requiring thread  $t_0$ 's collaboration after  $t_0$  executes the exit protocol



# Peterson's algorithm is starvation free

(No thread keeps waiting to enter the critical section)

<pre> boolean[] enter = {false, false};    int yield = 0    1; </pre> <hr/> <table border="0"> <tr> <td style="width: 50%; vertical-align: top;"> <b>thread <math>t_0</math></b> <pre> 1 while (true) { 2   // entry protocol 3   enter[0] = true; 4   yield = 0; 5   await (!enter[1] 6       yield != 0); 7   critical section { ... } 8   // exit protocol 9   enter[0] = false; } </pre> </td> <td style="width: 50%; vertical-align: top;"> <b>thread <math>t_1</math></b> <pre> 10 11 12 13 14 15 16 17 18 </pre> <pre> while (true) {   // entry protocol   enter[1] = true;   yield = 1;   await (!enter[0]       yield != 1);   critical section { ... }   // exit protocol   enter[1] = false; } </pre> </td> </tr> </table>	<b>thread <math>t_0</math></b> <pre> 1 while (true) { 2   // entry protocol 3   enter[0] = true; 4   yield = 0; 5   await (!enter[1] 6       yield != 0); 7   critical section { ... } 8   // exit protocol 9   enter[0] = false; } </pre>	<b>thread <math>t_1</math></b> <pre> 10 11 12 13 14 15 16 17 18 </pre> <pre> while (true) {   // entry protocol   enter[1] = true;   yield = 1;   await (!enter[0]       yield != 1);   critical section { ... }   // exit protocol   enter[1] = false; } </pre>
<b>thread <math>t_0</math></b> <pre> 1 while (true) { 2   // entry protocol 3   enter[0] = true; 4   yield = 0; 5   await (!enter[1] 6       yield != 0); 7   critical section { ... } 8   // exit protocol 9   enter[0] = false; } </pre>	<b>thread <math>t_1</math></b> <pre> 10 11 12 13 14 15 16 17 18 </pre> <pre> while (true) {   // entry protocol   enter[1] = true;   yield = 1;   await (!enter[0]       yield != 1);   critical section { ... }   // exit protocol   enter[1] = false; } </pre>	

# Peterson's algorithm satisfies mutual exclusion

*Instead of building the state/transition diagram, we can also prove mutual exclusion by contradiction:*

- Assume  $t_0$  and  $t_1$  both are in their critical section
- We have  $\text{enter}[0] == \text{true}$  and  $\text{enter}[1] == \text{true}$  ( $t_0$  and  $t_1$  set them before last entering their critical sections)
- Either  $\text{yield} == 0$  or  $\text{yield} == 1$   
Without loss of generality, assume  $\text{yield} == 0$
- Before last entering its critical section,  $t_0$  must have set  $\text{yield}$  to 0; after that it cannot have changed  $\text{yield}$  again
- To enter its critical section,  $t_0$  must have read  $\text{yield} == 1$  (since  $\text{enter}[1] == \text{true}$ ), so  $t_1$  must have set  $\text{yield}$  to 1 after  $t_0$  last changed  $\text{yield}$  to 0
- Since neither thread can have changed  $\text{yield}$  to 0 after that, we must have  $\text{yield} == 1$

```

boolean[] enter = {false, false};    int yield = 0 || 1;
                                         thread t0           thread t1
                                         10
1  while (true) {                      while (true) {
2   // entry protocol                  // entry protocol
3   enter[0] = true;                  enter[1] = true;
4   yield = 0;                         yield = 1;
5   await (!enter[1])                await (!enter[0])
                                         ||                                ||
                                         yield != 0);                  yield != 1);
6   critical section { ... }          critical section { ... }
7   // exit protocol                  // exit protocol
8   enter[0] = false;                 enter[1] = false;
9  }                                  }
                                         11
                                         12
                                         13
                                         14
                                         15
                                         16
                                         17
                                         18
  
```

**Contradiction!**

# Peterson's algorithm is starvation free

Suppose  $t_0$  is waiting to enter its critical section. At the same time,  $t_1$  must be doing one of four things:

1.  $t_1$  is in its critical section: then, it will eventually leave it;
2.  $t_1$  is in its non-critical section: then, `enter[1] == false`, so  $t_0$  can enter its critical section;
3.  $t_1$  is waiting to enter its critical section: then, `yield` is either 0 or 1, so one thread can enter the critical section;
4.  $t_1$  keeps on entering and exiting its critical section: this is impossible because after  $t_1$  sets `yield` to 1 it cannot cycle until  $t_0$  has a chance to enter its critical section (and reset `yield`).

In all possible cases,  $t_0$  eventually gets a chance to enter the critical section, so there is no starvation

Since starvation freedom implies deadlock freedom:

Peterson's algorithm is a correct mutual exclusion protocol

# Peterson's algorithm is starvation free

... then thread  $t_0$   
will **NOT** starve  
(it can go in into the  
critical section since  
 $\text{enter}[1]=\text{false}$ )

<b>boolean[]</b> enter = { <b>false</b> , <b>false</b> }; <b>int</b> yield = 0    1;	
thread <i>t</i> <sub>0</sub>	thread <i>t</i> <sub>1</sub>
<pre>1 while (true) { 2   // entry protocol 3   enter[0] = <b>true</b>; 4   yield = 0; 5   await (!enter[1]                    yield != 0); 6   critical section { ... } 7   // exit protocol 8   enter[0] = <b>false</b>; 9 }</pre>	<pre>while (true) {   // entry protocol   enter[1] = <b>true</b>;   yield = 1;   await (!enter[0]                  yield != 1);   critical section { ... }   // exit protocol   enter[1] = <b>false</b>; }</pre>

If `yield=0` (or 1)  
and thread  $t_1$  stops  
executing here  
(before the entry  
protocol)...

# Peterson's algorithm for $n$ threads

Peterson's algorithm easily generalizes to  $n$  threads

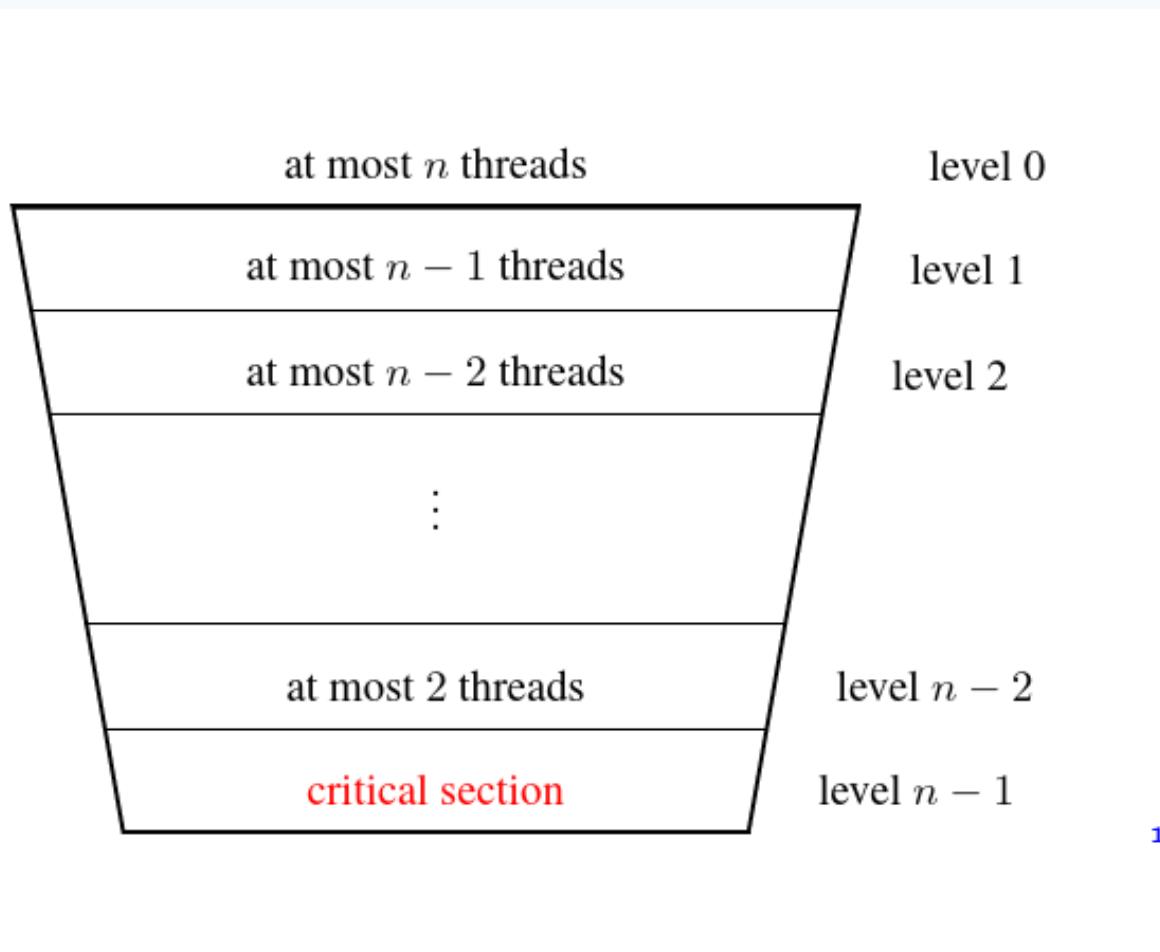
```
int[] enter = new int[n]; // n elements, initially all 0s
int[] yield = new int[n]; // use n - 1 elements 1..n-1


---


          thread x

1  while (true) {
2    // entry protocol
3    for (int i = 1; i < n; i++) {
4      enter[x] = i;    // want to enter level i
5      yield[i] = x;   // but yield first
6      await (∀ t != x: enter[t] < i
              || yield[i] != x);
                    wait until all other
                    threads are in lower levels
7    }
8    critical section { ... }
9    // exit protocol
10   enter[x] = 0; // go back to level 0
                    or another thread
                    is yielding
```

# Peterson's algorithm for $n$ threads



```

int[] enter = new int[n]; // n elements, initially all 0s
int[] yield = new int[n]; // use n - 1 elements 1..n-1

```

---

thread  $x$

```

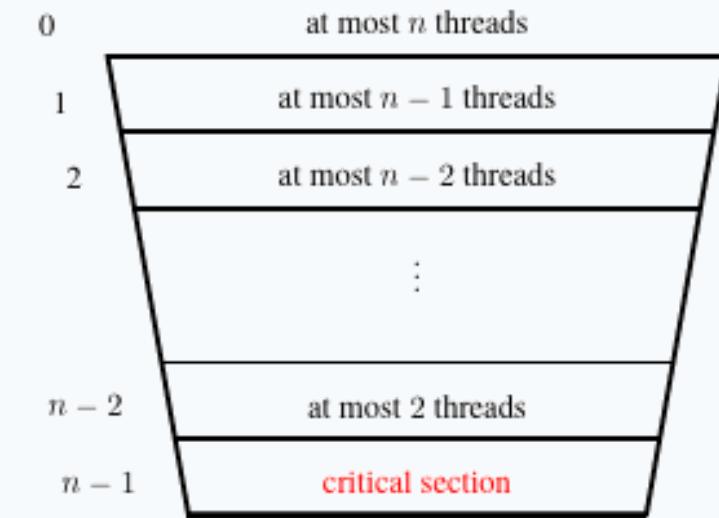
1  while (true) {
2    // entry protocol
3    for (int i = 1; i < n; i++) {
4      enter[x] = i;    // want to enter level i
5      yield[i] = x;   // but yield first
6      await (∀ t != x: enter[t] < i
7          || yield[i] != x);
8    }
9    critical section { ... }
10   // exit protocol
11   enter[x] = 0; // go back to level 0

```

# Peterson's algorithm for $n$ threads

Every thread goes through  $n - 1$  levels to enter the critical section:

- when a thread is at level 0 it is outside the entry region;
- when a thread is at level  $n - 1$  it is in the critical section;
- Thread  $x$  is in level  $i$  when it has finished the loop at line 6 with `enter[x] = i;`
- `yield[ℓ]` indicates the *last* thread that wants to enter level  $\ell$ ;
- to enter the next level, **wait until** there are no processes in higher levels, or another process (which entered the current level last) is yielding;
- **mutual exclusion**: at most  $n - \ell$  processes are in level  $\ell$ , thus at most  $n - (n - 1) = 1$  processes in critical section.




---

```
int[] enter = new int[n]; // n elements, initially all 0s
int[] yield = new int[n]; // use n - 1 elements 1..n-1
```

thread  $x$

```

1 while (true) {
2   // entry protocol
3   for (int i = 1; i < n; i++) {
4     enter[x] = i;    // want to enter level i
5     yield[i] = x;   // but yield first
6     await (∀ t != x: enter[t] < i
7       || yield[i] != x);
8   }
9   critical section { ... }
10  // exit protocol
11  enter[x] = 0; // go back to level 0

```

# Mutual exclusion with bounded waiting

# Bounded waiting (also called bounded bypass)

Peterson's algorithm guarantees freedom from starvation, but threads may get access to their critical section before other "older" threads

To describe this, we introduce more precise **properties of fairness**:

**Finite waiting (starvation freedom):** when a thread  $t$  is waiting to enter its critical section, it will **eventually** enter it

**Bounded waiting:** when a thread  $t$  is waiting to enter its critical section, the maximum number of times other arriving threads are allowed to enter their critical section before  $t$  is **bounded** by a function of the number of contending threads

**$r$ -bounded waiting:** when a thread  $t$  is waiting to enter its critical section, the maximum number of times other arriving threads are allowed to enter their critical section before  $t$  is less than  $r + 1$

**First-come-first-served:** **0-bounded** waiting

# The Bakery algorithm

Lamport's Bakery algorithm achieves mutual exclusion, deadlock freedom, and first-come-first-served access

It is based on the idea of waiting threads getting a ticket number:

- Because of lack of atomicity, two threads may end up with the same ticket number
- In that case, their thread identifier number is used to force an order
- The tricky part is evaluating multiple variables (the ticket numbers of all other waiting processes) consistently
- Idea: a thread raises a flag when computing the number; other threads then wait to compute the numbers

Main drawback (compared to Peterson's algorithm): the original version of the Bakery algorithm may use arbitrarily large integers (the ticket numbers) in shared variables

# Implementing mutual exclusion algorithms in Java

Now that you know how to do it...

... don't do it!

Learning how to achieve mutual exclusion using only atomic reads and writes  
[has educational value](#), but you should not use it in realistic programs

- Use the locks and semaphores available in Java's [standard library](#)
- We will still give an overview of the things to know if you were to implement Peterson's algorithm, and similar ones, [from the ground up](#)

# Peterson's lock in Java: 2 threads

```

class PetersonLock implements Lock {
    private volatile boolean enter0 = false, enter1 = false;
    private volatile int yield;

    public void lock()
    {   int me = getThreadId();
        if (me == 0) enter0 = true;
        else enter1 = true;
        yield = me;
        while ((me == 0) ? (enter1 && yield == 0)
                  : (enter0 && yield == 1)) {}  }
    }

    public void unlock()
    {   int me = getThreadId();
        if (me == 0) enter0 = false;
        else enter1 = false;  }

    private volatile long id0 = 0;
  
```

**volatile** is required  
 for correctness

The loop will exit:  
 if me=0 and (enter1 is false or yield is 1)  
 or  
 if me=1 and (enter0 is false or yield is 0)

# Instruction execution order

When we designed and analyzed concurrent algorithms, we implicitly assumed that threads execute instructions in textual program order

This is **not guaranteed** by the Java language – or, for that matter, by most programming languages – when threads access shared fields

(Read “The silently shifting semicolon” <http://drops.dagstuhl.de/opus/volltexte/2015/5025/> for a nice description of the problems)

- **Compilers** may reorder instructions based on static analysis, which does not know about threads.
- **Processors** may delay the effect of writes when the cache is committed to memory

This adds to the complications of writing low-level concurrent software correctly



# Instruction execution order

The compiler might  
Decide to move this instruction



```
class PetersonLock implements Lock {  
    private volatile boolean enter0 = false, enter1 = false;  
    private volatile int yield;  
  
    public void lock()  
    {  
        int me = getThreadId();  
        if (me == 0) enter0 = true;  
        else enter1 = true;  
        yield = me;  
        while ((me == 0) ? (enter1 && yield == 0)  
               : (enter0 && yield == 1)) {} }  
  
    public void unlock()  
    {  
        int me = getThreadId();  
        if (me == 0) enter0 = false;  
        else enter1 = false; }  
  
    private volatile long id0 = 0;
```

- **Compilers** may reorder instructions based on static analysis, which does not know about threads.
- **Processors** may delay the effect of writes when the cache is committed to memory

This adds to the complications of writing low-level concurrent software correctly



# Volatile fields

Accessing a field (attribute) declared as **volatile** forces synchronization, and thus prevents optimizations from reordering instructions in a way that alters the “**happens before**” relationship defined by a program’s textual order

- By using **volatile** we ensure the variable changes at runtime and that the compiler should not cache its value for any reason

When accessing a shared variable that is accessed concurrently:

- declare the variable as **volatile**
- or guard access to the variable with **locks** (or other synchronization primitives)

# Arrays and **volatile**

Java does **not support** arrays *whose elements are volatile*

That's why we used two scalar **boolean** var when implementing Peterson's lock

## Workarounds:

- Use an object of class `AtomicIntegerArray` in package `java.util.concurrent.atomic` which guarantees atomicity of accesses to its elements (the field itself need not be declared volatile)
- Make sure that there is a read to a **volatile** field before every read to elements of the shared array, and that there is a write to a **volatile** field after every write to elements of the shared array; this forces synchronization indirectly (may be tricky to do correctly!)
- **Explicitly** guard accesses to shared arrays with a **lock**: this is the high-level solution which we will preferably use

# Peterson's lock in Java: 2 threads, with atomic arrays

```
class PetersonAtomicLock implements Lock {  
    private AtomicIntegerArray enter = new AtomicIntegerArray(2);  
    private volatile int yield;  
  
    public void lock() {  
        int me = getThreadId();  
        int other = 1 - me;  
        enter.set(me, 1);  
        yield = me;  
        while (enter.get(other) == 1 && yield == me) {}  
    }  
  
    public void unlock() {  
        int me = getThreadId();  
        enter.set(me, 0);  
    }  
}
```

# Peterson's lock in Java: 2 threads

"Classic":

```
class PetersonLock implements Lock {
  private volatile boolean enter0 = false,
                      enter1 = false;
  private volatile int yield;

  public void lock()
  {   int me = getThreadId();
      if (me == 0) enter0 = true;
      else enter1 = true;
      yield = me;
      while ((me == 0) ? (enter1 && yield == 0)
                     : (enter0 && yield == 1)) {}}

  public void unlock()
  {   int me = getThreadId();
      if (me == 0) enter0 = false;
      else enter1 = false;
  }
}
```

With atomic arrays:

```
class PetersonAtomicLock implements Lock {
  private AtomicIntegerArray enter = new AtomicIntegerArray(2);
  private volatile int yield;

  public void lock()
  {   int me = getThreadId();
      int other = 1 - me;
      enter.set(me, 1);
      yield = me;
      while (enter.get(other) == 1
             && yield == me) {}

  public void unlock()
  {   int me = getThreadId();
      enter.set(me, 0);
  }
}
```

# Mutual exclusion needs $n$ memory locations

Peterson's algorithm for  $n$  threads uses  $\Theta(n)$  shared memory locations (two  $n$ -element arrays)

- One can prove that this is the minimum amount of shared memory needed to have mutual exclusion *if only atomic reads and writes* are available
- This is one reason why synchronization using only atomic reads and writes is impractical
- We need more powerful primitive operations:
  - atomic test-and-set operations
  - support for suspending and resuming threads explicitly

# Test-and-set

The **test-and-set** operation **boolean testAndSet()** works on a Boolean variable **b** as follows: **b.testAndSet()** **atomically** returns the current value of **b** and sets **b** to **true**

Java class `AtomicBoolean` implements test-and-set:

```
package java.util.concurrent.atomic;
public class AtomicBoolean {
    AtomicBoolean(boolean initialValue); // initialize to `initialValue'

    boolean get();                      // read current value
    void set(boolean newValue);         // write `newValue'

    // return current value and write `newValue'
    boolean getAndSet(boolean newValue);
        // testAndSet() is equivalent to getAndSet(true)
}
```

# A lock using test-and-set

An implementation of  $n$ -process mutual exclusion using a single Boolean variable with test-and-set and busy waiting:

```
public class TASLock implements Lock {  
    AtomicBoolean held =  
        new AtomicBoolean(false);  
  
    public void lock() {  
        while (held.getAndSet(true)) {  
            } // await (!testAndSet());  
    }  
  
    public void unlock() {  
        held.set(false); // held = false;  
    }  
}
```

- Variable `held` is true iff the lock is held by some thread
- When locking (executing `lock`):
  - as long as `held` is true (someone else holds the lock), keep resetting it to true and wait
  - as soon as `held` is false: leave the loop and `held` is set it to true
    - You hold the lock now
- When unlocking (executing `unlock`): set `held` to false

# A lock using test-and-test-and-set

A lock implementation using a single Boolean variable with test-and-test-and-set and busy waiting:

```
public class TTASLock extends TASLock {  
    @Override  
    public void lock() {  
        while (true) {  
            while (held.get()) {}  
            if (!held.getAndSet(true))  
                return;  
        }  
    }  
}
```

When locking (executing `lock`):

- spin until `held` is false
- then check if `held` is still false, and if it is set it to true (you hold the lock now), then return
- otherwise it means another thread “stole” the lock from you; then repeat the locking procedure from the beginning

This variant tends to *perform better*, since the busy waiting is local to the cached copy as long as no other thread changes the lock’s state (Read section 7.2 of Herlihy and Shavit book)

# Implementing semaphores

# Semaphores: recap

A (general/counting) **semaphore** is a data structure with interface:

```
interface Semaphore {  
    int count();      // current value of counter  
    void up();        // increments counter  
    void down();     // decrements counter  
}
```

Several threads share the same object `sem` of type `Semaphore`:

- initially `count` is set to a nonnegative value `C` (the initial **capacity**)
- a call to `sem.up()` **atomically increments** `count` by one
- a call to `sem.down()`: **waits** until `count` is positive, and then **atomically decrements** `count` by one

# Semaphores with locks

An implementation of semaphores using locks and busy waiting:

```

class SemaphoreBusy implements Semaphore {
  private int count;

  public synchronized void up() {
    count = count + 1;
  }

  public void down() {
    while (true) {
      synchronized (this) {
        if (count > 0) { // await (count > 0);
          count = count - 1; return;
        }
      }
    }
  }

  public synchronized int count() {
    return count;
  }
}

```

Executed exclusively

Why not lock the whole method?

To avoid blocking other threads to enter the method  
(avoid that the first thread calling down is the first to get the lock!)

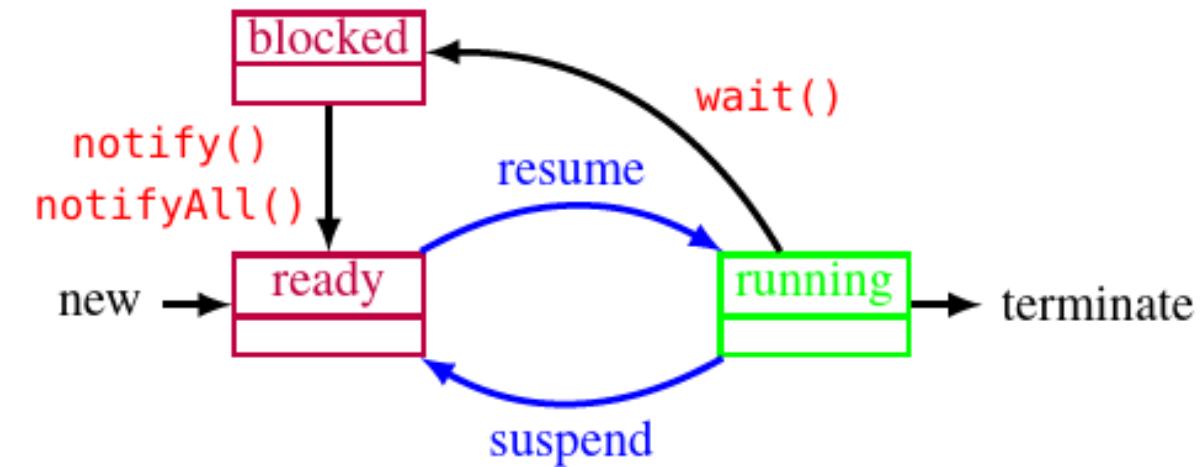
Does this have to be **synchronized**?  
Yes, if count is not **volatile**

# Suspending and resuming threads

To avoid **busy waiting**, we have to rely on more powerful synchronization primitives than only reading and writing variables

A standard solution uses Java's **explicit scheduling of threads**

- calling `wait()` suspends the currently running thread
- calling `notify()` moves one (nondeterministically chosen) blocked thread to the **ready** state
- calling `notifyAll()` moves all blocked threads to the **ready** state



Waiting and notifying only affects the threads that are locked on the **same shared object** (using **synchronized** blocks or methods)

# Weak semaphores with suspend/resume

An implementation of **weak** semaphores using `wait()` and `notify()`

```

class SemaphoreWeak implements Semaphore {
    private int count;

    public synchronized void up() {
        count = count + 1;
        notify();           // wake up a waiting thread
    }

    public synchronized void down() throws InterruptedException {
        while (count == 0) wait();   // suspend running thread
        count = count - 1;          // now count > 0
    }

    public synchronized int count() {
        return count;
    }
}

```

Since `notify` is nondeterministic  
this is a **weak** semaphore

`wait` releases the object lock  
(so other threads can enter the method even if it is marked as “`synchronized`”)

In general, `wait` must be called in a loop in case of spurious wakeups;  
this is not busy waiting (and it's required by Java's implementation)

# Strong semaphores with suspend/resume

An implementation of **strong** semaphores using `wait()` and `notifyAll()`

```

class SemaphoreStrong implements Semaphore {
    public synchronized void up() {
        if (blocked.isEmpty())
            count = count + 1;
        else notifyAll();           // wake up all waiting threads
    }

    public synchronized void down() throws InterruptedException {
        Thread me = Thread.currentThread();
        blocked.add(me);          // enqueue me
        while (count == 0 || blocked.element() != me)
            wait();                // I'm enqueued when suspending
        // now count > 0 and it's my turn: dequeue me and decrement
        blocked.remove();         count = count - 1;
    }

    private final Queue<Thread> blocked = new LinkedList<>();
    private int count;
  
```

Check there are no suspended threads

**Wrong!**

Keeps suspending the thread if the count is 0 or I am not the first in the queue

# Strong semaphores with suspend/resume

An implementation of **strong** semaphores using `wait()` and `notifyAll()`

```
class SemaphoreStrong implements Semaphore {    Removed if (blocked.isEmpty())  
  
    public synchronized void up() {  
        count = count + 1;  
        notifyAll();      // wake up all waiting threads  
    }  
  
    public synchronized void down() throws InterruptedException {  
        Thread me = Thread.currentThread();  
        blocked.add(me); // enqueue me  
        while (count == 0 || blocked.element() != me)  
            wait();        // I'm enqueued when suspending  
        // now count > 0 and it's my turn: dequeue me and decrement  
        blocked.remove(); count = count - 1;  
    }  
  
    private final Queue<Thread> blocked = new LinkedList<>();  
  
    private int count;  
}
```





Debugging concurrent  
programs is very  
difficult!

# General semaphores using binary semaphores

A general semaphore can be implemented using just **two binary semaphores**

Barz's solution in pseudocode (with  $\text{capacity} > 0$ ):

```
BinarySemaphore mutex = 1; // protects access to count
BinarySemaphore delay = 1; // blocks threads in down until count > 0
int count = capacity; // value of general semaphore
void up()
{ mutex.down(); // get exclusive access to count
  count = count + 1; // increment count
  if (count == 1) delay.up(); // release threads blocking on down
  mutex.up(); } // release exclusive access to count
void down()
{ delay.down(); // block other threads starting down
  mutex.down(); // get exclusive access to count
  count = count - 1; // decrement count
  if (count > 0) delay.up(); // release threads blocking on down
  mutex.up(); } // release exclusive access to count
```

© 2016–2019 Carlo A. Furia, Sandro Stucki



Except where otherwise noted, this work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

To view a copy of this license, visit

<http://creativecommons.org/licenses/by-sa/4.0/>.

```
1 class SemaphoreStrong implements Semaphore {
2     public synchronized void up()
3     {   if (blocked.isEmpty()) count = count + 1;
4         else notifyAll();    } // wake up all waiting threads
5
6     public synchronized void down() throws InterruptedException
7     {   Thread me = Thread.currentThread();
8         blocked.add(me); // enqueue me
9         while (count == 0 || blocked.element() != me)
10             wait();          // I'm enqueued when suspending
11         // now count > 0 and it's my turn: dequeue me and decrement
12         blocked.remove(); count = count - 1;    }
13
14     private final Queue<Thread> blocked = new LinkedList<>();
```

```
15 class StrongSemUser implements Runnable {  
16     private SemaphoreStrong sem = new SemaphoreStrong(1);  
17  
18     public void run()  
19     {    while (true) {  
20             // Non critical  
21             sem.down();  
22             // Critical  
23             sem.up();  
24         }  
25     }
```

```
class StrongSemUser implements Runnable {  
    private SemaphoreStrong sem = new SemaphoreString(1);  
  
    public void run()  
    {    while (true) {  
        // Non critical  
        sem.down();  
        // Critical  
        sem.up();  
    }  
}
```

## Peterson's algorithm

Combine the ideas behind the second and third attempts:

- thread  $t_k$  first sets `enter[k]` to true
- but lets the other thread go first – by setting `yield`

```
boolean[] enter = {false, false}; int yield = 0 || 1;

Equivalent to:
wait while
(enter[1]==true
 &
 yield==0)

Enter only when
(enter[1]==false
 OR
 yield==1)

    thread t0           thread t1
    1 while (true) {      1 while (true) {
    2   // entry protocol 2   // entry protocol
    3   enter[0] = true; 3   enter[1] = true;
    4   yield = 0;        4   yield = 1;
    5   await (enter[1]); 5   await (enter[0]);
    6   yield = 0;        6   yield = 1;
    7   // exit protocol 7   // exit protocol
    8   enter[0] = false; 8   enter[1] = false;
    9 }                   9 }
```

Peterson's algorithm for  $n$  threads

Peterson's algorithm easily generalizes to  $n$  threads

```
int[] enter = new int[n]; // n elements, initially all 0s
int[] yield = new int[n]; // use n - 1 elements 1..n-1

thread x

1 while (true) {
2   // entry protocol
3   for (int i = 1; i < n; i++) {
4     enter[i] = i; // want to enter level i
5     yield[i] = i; // but yield first
6     await (yield[i] >= x);
7     yield[i] = i;
8   }
9 critical section { ... }
10 // exit protocol
11 enter[x] = false;
12 yield[x] = 0;
```

33



## Peterson's lock in Java: 2 threads

```
class PetersonLock implements Lock {
  private volatile boolean enter0 = false, enter1 = false;
  private volatile int yield;

  public void lock()
  {
    int me = getThreadId();
    if (me == 0) enter0 = true;
    else enter1 = true;
    yield = me;
    while ((me == 0) ? (enter1 && yield == 0)
                  : (enter0 && yield == 1)) {} // The loop will exit:
    if me==0 and (enter1 is false or yield is 1)
    or
    if me==1 and (enter0 is false or yield is 0)
  }

  private volatile long id0 = 0;
```

## Instruction execution order

When we designed and analyzed concurrent algorithms, we implicitly assumed that threads *execute instructions in textual program order*

This is **not guaranteed** by the Java language – or, for that matter, by most programming languages – when threads access shared fields

(Read "The silently shifting semicolon" <http://drops.dagstuhl.de/opus/volltexte/2015/5025/> for a nice description of the problems)

- Compilers may reorder instructions based on static analysis, which does not know about threads.
- Processors may delay the effect of writes when the cache is committed to memory

This adds to the complications of writing low-level concurrent software correctly



52

## Test-and-set

The **test-and-set** operation `boolean testAndSet()` works on a Boolean variable `b` as follows: `b.testAndSet()` atomically returns the current value of `b` and sets `b` to `true`

Java class `AtomicBoolean` implements test-and-set:

```
package java.util.concurrent.atomic;
public class AtomicBoolean {

  AtomicBoolean(boolean initialValue); // initialize to 'initialValue'

  boolean get(); // read current value
  void set(boolean newValue); // write 'newValue'

  // return current value and write 'newValue'
  boolean getAndSet(boolean newValue);
    // testAndSet() is equivalent to getAndSet(true)
}
```

53

```
class PetersonAtomicLock implements Lock {
  private AtomicIntegerArray enter = new AtomicIntegerArray(2);
  private volatile int yield;

  public void lock() {
    int me = getThreadId();
    int other = 1 - me;
    enter.set(me, 1);
    yield = me;
    while (enter.get(other) == 1 && yield == me) {}
  }

  public void unlock() {
    int me = getThreadId();
    enter.set(me, 0);
  }
}
```

57

# Synchronization problems with semaphores

Lecture 4 of TDA384/DIT391

Principles of Concurrent Programming

Nir Piterman and Gerardo Schneider

Chalmers University of Technology | University of Gothenburg



UNIVERSITY OF  
GOTHENBURG



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY



# Lesson's menu

Dining philosophers

- Dining philosophers

Producer-consumer

- Producer-consumer

- Barriers

Barriers

- Readers-writers

Readers-writers

# Lesson's menu

- Dining philosophers
- Producer-consumer
- Barriers
- Readers-writers
  - Identify problems of synchronization
  - What issues and problems can arise
  - Patterns for introducing synchronization

## Learning outcomes

### *Knowledge and understanding:*

- demonstrate knowledge of the issues and problems that arise in writing correct concurrent programs;
- identify the problems of synchronization typical of concurrent programs, such as race conditions and mutual exclusion

### *Skills and abilities:*

- apply common patterns, such as lock, semaphores, and message-passing synchronization for solving concurrent program problems;
- apply practical knowledge of the programming constructs and techniques offered by modern concurrent programming languages;
- implement solutions using common patterns in modern programming languages

### *Judgment and approach:*

- evaluate the correctness, clarity, and efficiency of different solutions to concurrent programming problems;
- judge whether a program, a library, or a data structure is safe for usage in a concurrent setting;
- pick the right language constructs for solving synchronization and communication problems between computational units.

# A gallery of synchronization problems

- Today we go through several **classical synchronization problems** and solve them using threads and semaphores
- If you want to learn about many other synchronization problems and their solutions
  - "The little book of semaphores" by A. B. Downey: <http://greenteapress.com/sema...>
- We use **pseudo-code** to simplify the details of Java syntax and libraries but which can be turned into fully functioning code by adding boilerplate
  - On the course website: can download fully working implementations of some of the problems
- Recall that we occasionally annotate classes with *invariants* using the pseudo-code keyword **invariant**
  - **Not** a valid Java keyword – that is why we highlight it in a different color – but we will use it to help make more explicit the behavior of classes
  - We also use **at(i)** or **at(i, j)** to indicate the **number of threads** that are at location **i** or between locations **i, j**. (That's not Java either)

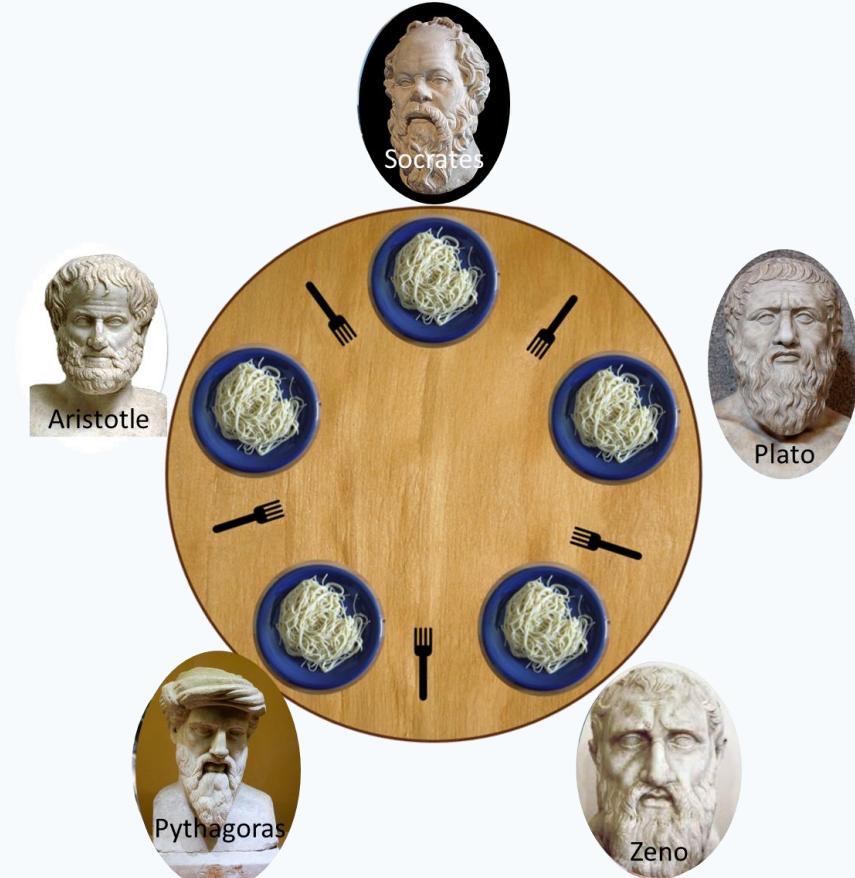
# Dining philosophers

# The dining philosophers (reminder)

The **dining philosophers** is a classic synchronization problem introduced by Dijkstra

It illustrates the problem of deadlocks using a colorful metaphor (by Hoare)

- Five philosophers are sitting around a dinner table, with a fork in between each pair of adjacent philosophers
- Each philosopher alternates between thinking (**non-critical section**) and eating (**critical section**)
- In order to eat, a philosopher needs to pick up the **two forks** that lie to the philosopher's left and right
- Since the forks are **shared**, there is a **synchronization** problem between philosophers (**threads**)



# Dining philosophers: the problem

```
interface Table {  
  
    // philosopher k picks up forks  
  
    void getForks(int k);  
  
    // philosopher k releases forks  
  
    void putForks(int k);  
  
}
```

Properties of a good solution:

- support an arbitrary number of philosophers
- deadlock freedom
- starvation freedom
- reasonable efficiency: eating in parallel still possible

**Dining philosophers' problem:** implement `Table` such that:

- forks are held exclusively by one philosopher at a time
- each philosopher only accesses adjacent forks

# The philosophers

Each philosopher continuously alternate between thinking and eating; the table must **guarantee** proper **synchronization** when eating

```
Table table; // table shared by all philosophers

---

philosopherk  
  
while (true) {  
    think(); // think  
    table.getForks(k); // wait for forks  
    eat(); // eat  
    table.putForks(k); // release forks  
}
```

# Left and right

For convenience, we introduce a consistent numbering scheme for forks and philosophers, in a way that it is easy to refer to the left or right fork of each philosopher.

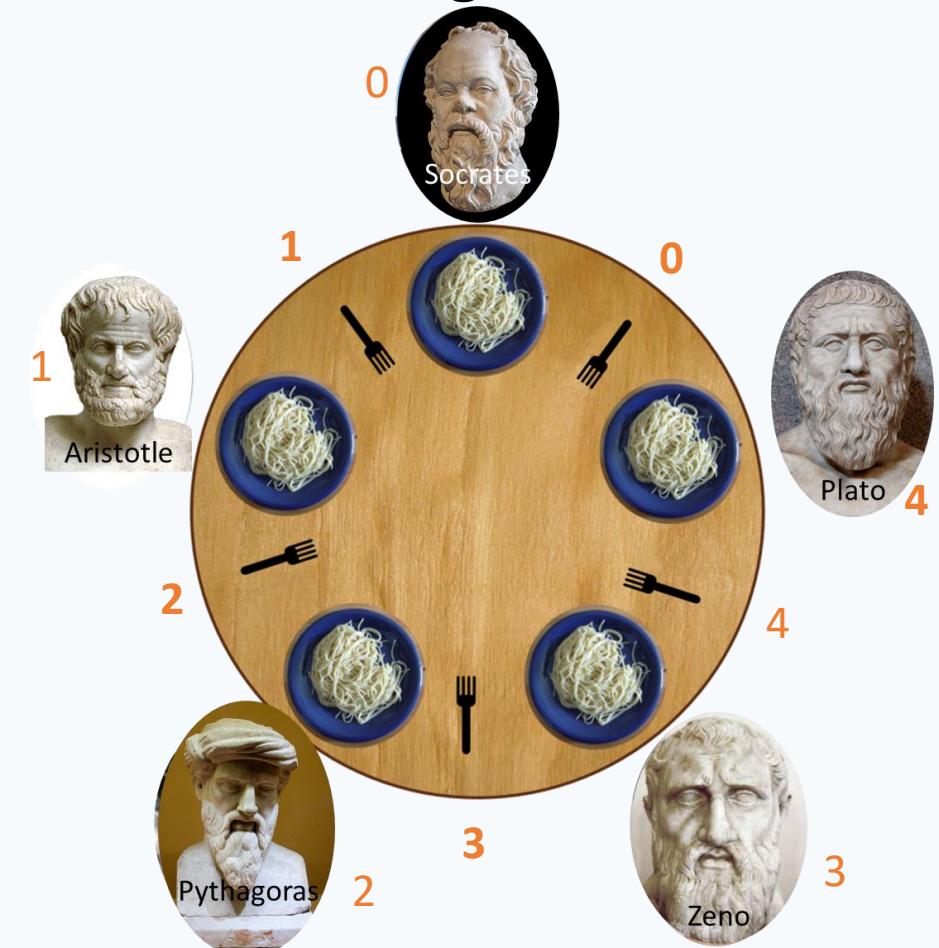
// in classes implementing Table:

// fork to the left of philosopher k

```
public int left(int k) {
    return k;
}
```

// fork to the right of philosopher k

```
public int right(int k) {
    // N is the number of philosophers
    return (k + 1) % N;
}
```



# Dining philosophers with locks and semaphores

- We use **semaphores** to enforce mutual exclusion when philosophers access the forks

First solution needs only **locks**:

```
Lock[] forks = new Lock[N]; // array of locks
```

- One lock per fork
- `forks[i].lock()` to pick up fork  $i$ :  
 $\text{forks}[i]$  is held if fork  $i$  is held
- `forks[i].unlock()` to put down fork  $i$ :  
 $\text{forks}[i]$  is available if fork  $i$  is available

# Dining philosophers with semaphores: first attempt

In the first attempt, every philosopher picks up the **left** fork **and then the right** fork:

```
public class DeadTable implements Table {  
    Lock[] forks = new Lock[N];  
  
    public void getForks(int k) {  
        // pick up left fork  
        forks[left(k)].lock();  
        // pick up right fork  
        forks[right(k)].lock();  
    }  
  
    public void putForks(int k) {  
        // put down left fork  
        forks[left(k)].unlock();  
        // put down right fork  
        forks[right(k)].unlock();  
    }  
}
```

All forks initially available

# Dining philosophers with semaphores: first attempt

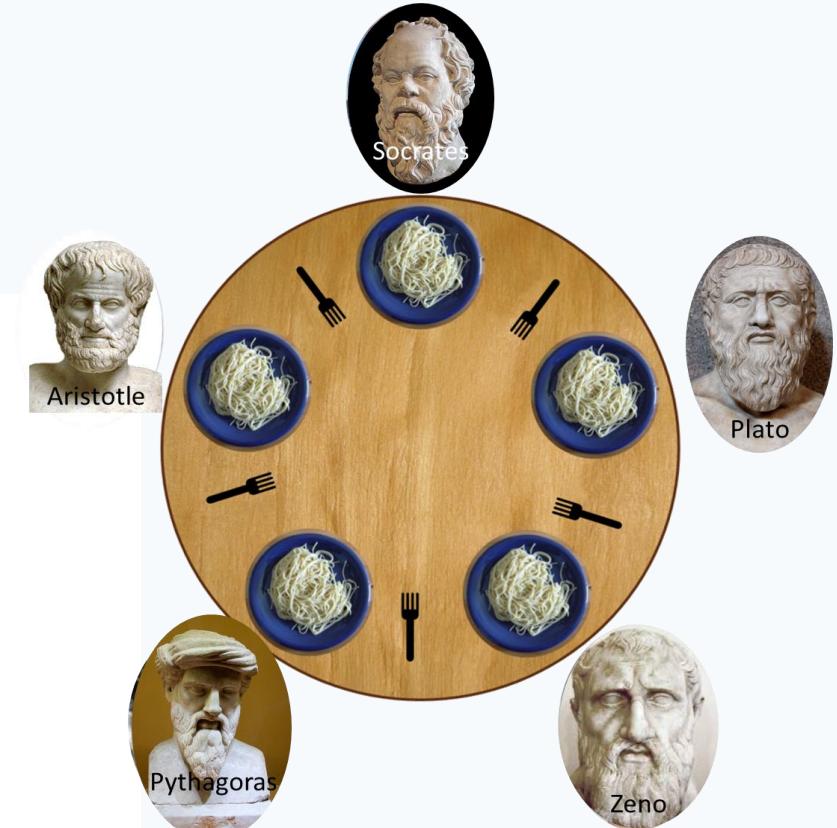
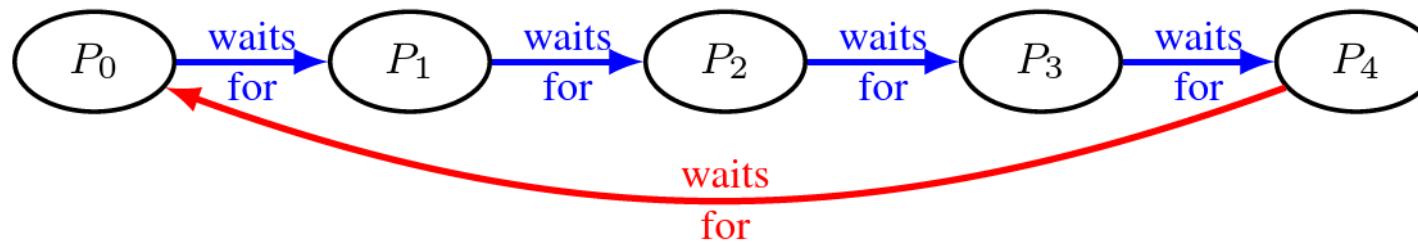
```

public class DeadTable implements Table
{
    Lock[] forks = new Lock[N];

    public void getForks(int k) {
        // pick up left fork
        forks[left(k)].lock();
        // pick up right fork
        forks[right(k)].lock();
    }
}
  
```

if all philosophers hold  
left fork: deadlock!

A **deadlock** may occur because of **circular waiting**:

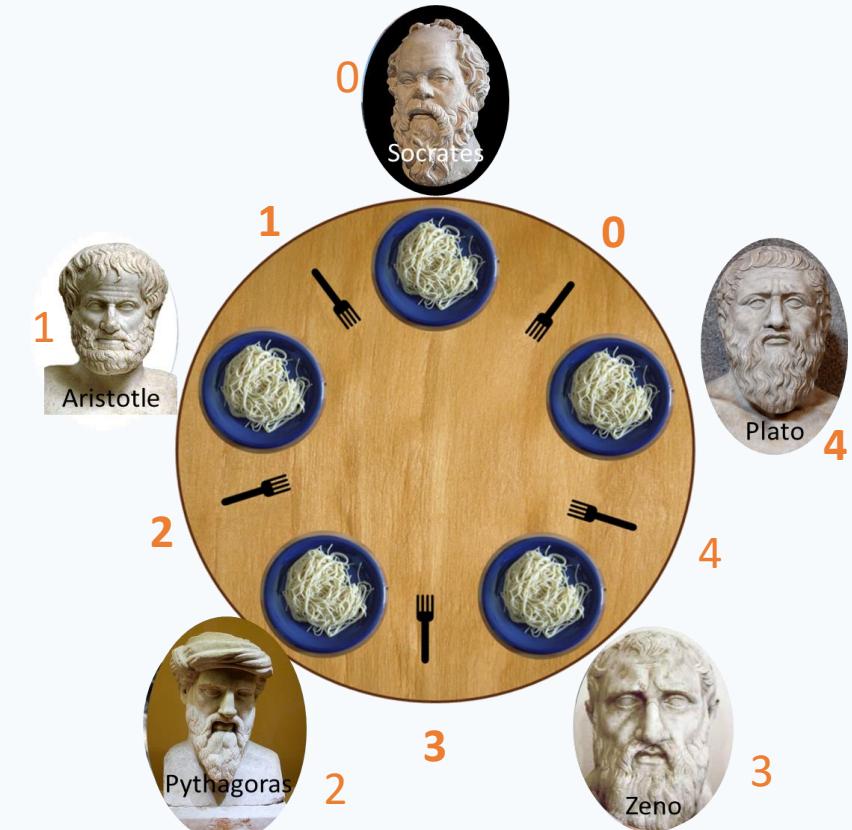


# Dining philosophers solution 1: breaking the symmetry

Having one philosopher pick up forks in a **different order** than the others is sufficient to break the symmetry, and thus to avoid deadlock

```
public class AsymmetricTable implements Table {
    Lock[] forks = new Lock[N];

    public void getForks(int k) {
        if (k == N) { // right before left
            forks[right(k)].lock();
            forks[left(k)].lock();
        } else { // left before right
            forks[left(k)].lock();
            forks[right(k)].lock();
        }
    }
    // putForks as in DeadTable
}
```



## Breaking symmetry to avoid deadlock

**Breaking the symmetry** is a **general strategy to avoid deadlock** when acquiring multiple shared resources:

- assign a **total order** between the shared resources  $R_0 < R_1 < \dots < R_M$
- a thread can try to obtain resource  $R_i$ , with  $i > j$ , only **after** it has successfully obtained resource  $R_j$

Recall the *Coffman conditions* from Lecture 2...:

1. **mutual exclusion**: exclusive access to the shared resources
2. **hold and wait**: request one resource while holding another
3. **no preemption**: resources cannot forcibly be released
4. **circular wait**: threads form a circular chain, each waiting for a resource the next is holding

Circular wait is a necessary condition for a deadlock to occur

# Dining philosophers solution 2: bounding resources

Limiting the number of philosophers active at the table to  $M < N$  ensures that there are enough resources for everyone at the table, thus avoiding deadlock

```

public class SeatingTable implements Table {
    Lock[] forks = new Lock[N];
    Semaphore seats = new Semaphore(M); // # available seats

    public void getForks(int k) {
        // get a seat
        seats.down();
        // pick up left fork
        forks[left(k)].lock();
        // pick up right fork
        forks[right(k)].lock();
    }

    public void putForks(int k) {
        // put down left fork
        forks[left(k)].unlock();
        // put down right fork
        forks[right(k)].unlock();
        // leave seat
        seats.up();
    }
}
  
```

# Starvation-free philosophers

The two solutions to the dining philosophers problem also guarantee **freedom from starvation**, under the assumption that locks/semaphores (and scheduling) are fair

In the **asymmetric solution** (`AsymmetricTable`):

- if a philosopher  $P$  waits for a fork  $k$ ,  $P$  gets the fork as soon as  $P$ 's neighbor holding fork  $k$  releases it,
- $P$ 's neighbor eventually releases fork  $k$  because there are no deadlocks.

In the **bounded-resource solution** (`SeatingTable`):

- at most  $M$  philosophers are active at the table,
- the other  $N-M$  philosophers are waiting on `seats.down()`,
- the first of the  $M$  philosophers that finishes eating releases a seat,
- the philosopher  $P$  that has been waiting on `seats.down()` proceeds,
- similarly to the asymmetric solution,  $P$  also eventually gets the forks.

# Producer-consumer

# Producer-consumer: overview

Producers and consumer exchange items through a **shared buffer**:

- **producers** asynchronously produce items and store them in buffer
- **consumers** asynchronously consume items after removing them from buffer



# Producer-consumer: The problem

**Producer-consumer** problem: implement **Buffer** such that:

- producers and consumers access the buffer in mutual exclusion
- consumers block when the buffer is empty
- producers block when the buffer is full (bounded buffer variant)

```
interface Buffer<T> {  
    // add item to buffer; block if full  
    void put(T item);  
  
    // remove item from buffer; block if empty  
    T get();  
  
    // number of items in buffer  
    int count();  
}
```

## Producer-consumer: Desired properties

**Producer-consumer** problem: implement **Buffer** such that:

- producers and consumers access the buffer in mutual exclusion
- consumers block when the buffer is empty
- producers block when the buffer is full (bounded buffer variant)

Other properties that a good solution should have:

- support an arbitrary number of producers and consumers
- deadlock freedom
- starvation freedom

# Producers and consumers

Producers and consumers **continuously** and **asynchronously** access the buffer, which must **guarantee** proper **synchronization**

```
Buffer<Item> buffer;
```

---

producer<sub>n</sub>

```
while (true) {  
    // create a new item  
    Item item = produce();  
    buffer.put(item);  
}
```

consumer<sub>m</sub>

```
while (true) {  
    Item item = buffer.get();  
    // do something with 'item'  
    consume(item);  
}
```

# Unbounded shared buffer

```

public class UnboundedBuffer<T> implements Buffer<T> {
    Lock lock = new Lock(); // for exclusive access to buffer
    Semaphore nItems = new Semaphore(0); // number of items in buffer
    Collection storage = ...; // any collection (list, set, ...)
    invariant { storage.count() == nItems.count() + at(5,15-17); }
}
  
```

Solution based on  
one lock and one  
semaphore

```

1 public void put(T item) {
2   lock.lock(); // lock
3   // store item
4   storage.add(item);
5   nItems.up(); // update nItems
6   lock.unlock(); // release
7 }
8
9 public int count() {
10  return nItems.count(); // locking here?
11 }
  
```

Signals to  
consumers waiting  
in **get** that they  
can proceed

```

12 public T get() {
13   // wait until nItems > 0
14   nItems.down();
15   lock.lock(); // lock
16   // retrieve item
17   T item = storage.remove();
18   lock.unlock(); // release
19   return item;
20 }
  
```

# Buffer: method put

Can we execute `up` after `unlock`?

```

1 public void put(T item) {
2   lock.lock(); // lock
3   // store item
4   storage.add(item);
5   nItems.up(); // update nItems
6   lock.unlock(); // release
7 }
8
9 public int count() {
10  return nItems.count(); // locking here?
11 }
```

Executing `up` after `unlock`:

- No effects on other threads executing `put`: they only wait for `lock`
- If a thread is waiting for `nItems > 0` in `get`: it does not have to wait again for `lock` just after it has been signaled to continue
- If a thread is waiting for the `lock` in `get`: it may return with the buffer in a (temporarily) inconsistent state (broken invariant, but benign because temporary)

# Executing up after unlock

```

1  public void put(T item) {
2      lock.lock();
3      storage.add(item);
4      lock.unlock(); // Line 4
5      nItems.up();
6  }

```

Different numbers than  
original program

Old invariant needs rewriting

OLD: **invariant** { storage.count()  
== nItems.count() + at(5, 15-17); }

# elements in buffer  
**invariant** {  
storage.count() ==  
nItems.count() + at(4, 9-10);  
}

Value of nItem  
(semaphore counter)

```

7  public T get() {
8      nItems.down();
9      lock.lock();
10     T item = storage.remove();
11     lock.unlock();
12     return item; Temporary breaking  
of the invariant
13 }

```

#	producer put	consumer get	SHARED
+1	pc <sub>t</sub> : 3	pc <sub>u</sub> : 8	nItems: 1 buffer: ⟨x⟩
+2	pc <sub>t</sub> : 3	pc <sub>u</sub> : 9	nItems: 0 buffer: ⟨x⟩
+3	pc <sub>t</sub> : 4	pc <sub>u</sub> : 9	nItems: 0 buffer: ⟨x, y⟩
+4	pc <sub>t</sub> : 5	pc <sub>u</sub> : 9	nItems: 0 buffer: ⟨x, y⟩
+5	pc <sub>t</sub> : 5	pc <sub>u</sub> : 10	nItems: 0 buffer: ⟨x, y⟩
+6	pc <sub>t</sub> : 5	pc <sub>u</sub> : 11	nItems: 0 buffer: ⟨y⟩
+7	pc <sub>t</sub> : 5	pc <sub>u</sub> : 12	nItems: 0 buffer: ⟨y⟩
+8	pc <sub>t</sub> : 5	done	nItems: 0 buffer: ⟨y⟩
+9	done	done	nItems: 1 buffer: ⟨y⟩

# Unbounded shared buffer

```

public class UnboundedBuffer<T> implements Buffer<T> {
    Lock lock = new Lock(); // for exclusive access to buffer
    Semaphore nItems = new Semaphore(0); // number of items in buffer
    Collection storage = ...; // any collection (list, set, ...)
    invariant { storage.count() == nItems.count() + at(5,15-17); }
}

1 public void put(T item) {
2     lock.lock(); // lock
3     // store item
4     storage.add(item);
5     nItems.up(); // update nItems
6     lock.unlock(); // release
7 }
8
9 public int count() {
10    return nItems.count(); // locking here?
11 }

```

```

12 public T get() {
13     // wait until nItems > 0
14     nItems.down();
15     lock.lock(); // lock
16     // retrieve item
17     T item =storage.remove();
18     lock.unlock(); // release
19     return item;
20 }

```

## Buffer: method get

What happens if another thread gets the lock just after the current threads has decremented the semaphore `nItems`?

- If the other thread is a **producer**, it doesn't matter: as soon as `get` resumes execution, there will be one element in storage to remove
- If the other thread is a **consumer**, it must have synchronized with the current thread on `nItems.down()`, and the order of removal of elements from the buffer doesn't matter

Can we execute `down` after `lock`?

```
12  public T get() {  
13      // wait until nItems > 0  
14      nItems.down();  
15      lock.lock(); // lock  
16      // retrieve item  
17      T item =storage.remove();  
18      lock.unlock(); // release  
19      return item;  
20  }
```

# Buffer: method get

Executing down after lock:

- If the buffer is empty when locking, there is a **deadlock!**
  - Will not succeed executing `down()` since the buffer is empty: it blocks!

```
12  public T get() {  
13      // wait until nItems > 0  
14      lock.lock(); // lock  
15      nItems.down();  
16      // retrieve item  
17      T item =storage.remove();  
18      lock.unlock(); // release  
19      return item;  
20 }
```

# Bounded shared buffer

Two semaphores

```

public class BoundedBuffer<T> implements Buffer<T> {
    Lock lock = new Lock(); // for exclusive access to buffer
    Semaphore nItems = new Semaphore(0); // # items in buffer
    Semaphore nFree = new Semaphore(N); // # free slots in buffer
    Collection storage = ...; // any collection (list, set, ...)
    invariant { storage.count() == nItems.count() +
        + at(6,13-15) == N - nFree.count() - at(4-6,16) ; }
  
```

Size of buffer

```

1 public void put(T item) {
2   // wait until nFree > 0
3   nFree.down();
4   lock.lock(); // lock
5   // store item
6   storage.add(item);
7   nItems.up(); // update nItems
8   lock.unlock(); // release
9 }
  
```

May deadlock  
if swapped

OK to swap

```

10 public T get() {
11   // wait until nItems > 0
12   nItems.down();
13   lock.lock(); // lock
14   // retrieve item
15   T item = storage.remove();
16   nFree.up(); // update nFree
17   lock.unlock(); // release
18   return item;
19 }
  
```

May deadlock  
if swapped

OK to swap

## Waiting on multiple conditions?

The operations offered by semaphores **do not support** waiting on **multiple conditions** (not empty and not full in our case) using **only** one semaphore

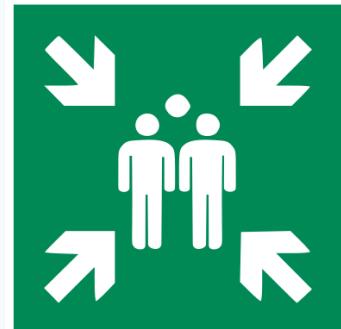
- Busy-waiting on the semaphore will **not work**:

```
// wait until there is space in the buffer
while (! (nItems.count() < N)) {};
// the buffer may be full again when locking!
lock.lock(); // lock
// store item
storage.add(item);
nItems.up(); // update nItems
lock.unlock(); // release
}
```

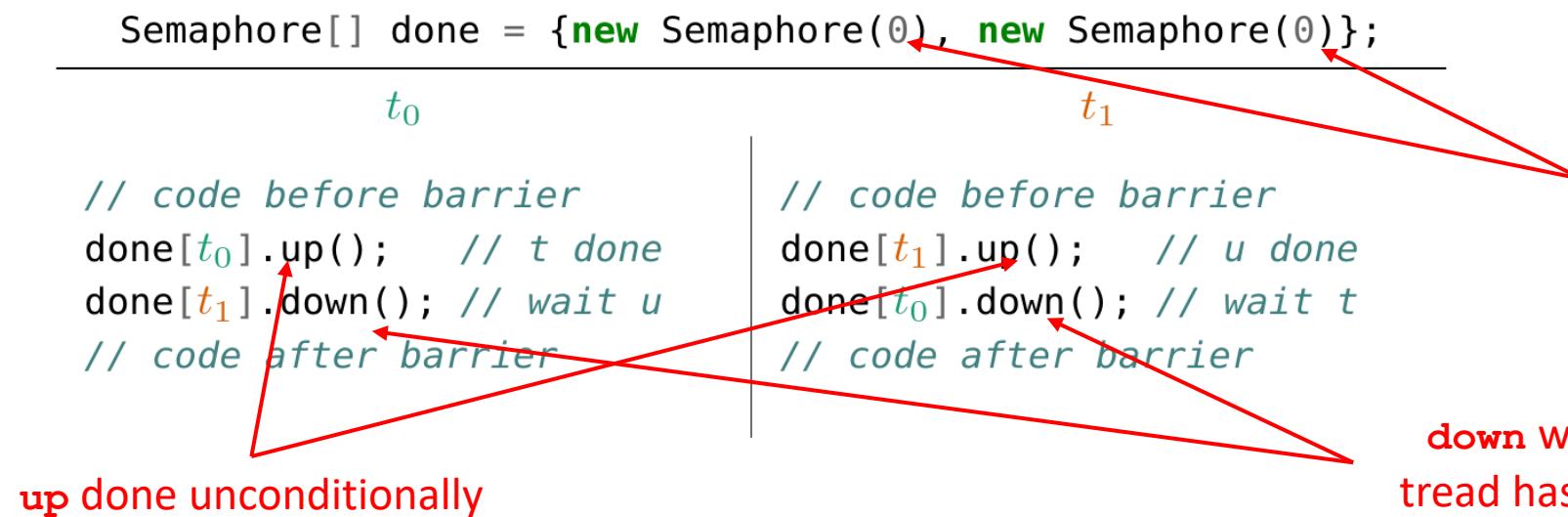
# Barriers

# Barriers (also called rendezvous)

A **barrier** is a form of synchronization where there is a point (the barrier) in a program's execution that **all threads** in a group have to reach **before** any of them is allowed to continue



A **solution** to the barrier synchronization problem **for 2 threads** with binary semaphores



Capacity 0 forces **up** before first **down**

**down** waits until the other thread has reaches the barrier

## Barriers: variant 1

The solution still works if  $t_0$  performs down before up – or, symmetrically, if  $t_1$  does the same

```
Semaphore[] done = new Semaphore(0), new Semaphore(0);
```

$t_0$

```
// code before barrier
done[t1].down(); // wait u
done[t0].up(); // t done
// code after barrier
```

$t_1$

```
// code before barrier
done[t1].up(); // u done
done[t0].down(); // wait t
// code after barrier
```

This is, however, a bit less efficient: the last thread to reach the barrier has to stop and yield to the other (one more context switch)

## Barriers: variant 2

The solution **deadlocks** if both  $t_0$  and  $t_1$  perform down before up

Deadlock

```
Semaphore[] done = new Semaphore(0), new Semaphore(0);
```

$t_0$

```
// code before barrier
done[t1].down(); // wait u
done[t0].up(); // t done
// code after barrier
```

$t_1$

```
// code before barrier
done[t0].down(); // wait t
done[t1].up(); // u done
// code after barrier
```

There is a **circular waiting**, because no thread has a chance to signal to the other that it has reached the barrier

# Barriers with $n$ threads (single use)

Keeping track of  $n$  threads reaching the barrier:

- nDone: number of threads that have reached the barrier
- lock: to update nDone atomically
- open: to release the waiting threads (“opening the barrier”)

```
int nDone = 0; // number of done threads
Lock lock = new Lock(); // mutual exclusion for nDone
Semaphore open = new Semaphore(0); // 1 iff barrier is open
```

---

thread  $t_k$

```
// code before barrier
lock.lock(); // lock nDone
nDone = nDone + 1; // I'm done
if (nDone == n) open.up(); // I'm the last: we can go!
lock.unlock(); // unlock nDone
open.down(); // proceed when possible
open.up(); // let the next one go
// code after barrier
```

Total number of expected threads

Can we switch these?

# Barriers with $n$ threads (single use): variant

```
int nDone = 0; // number of done threads
Lock lock = new Lock(); // mutual exclusion for nDone
Semaphore open = new Semaphore(0); // 1 iff barrier is open
```

thread  $t_k$

**Can we open the barrier after unlock?**

---

```
// code before barrier
lock.lock();           // lock nDone
nDone = nDone + 1;     // I'm done
lock.unlock();          // unlock nDone
if (nDone == n) open.up(); // I'm the last: we can go!
open.down();            // proceed when possible
open.up();              // let the next one go
// code after barrier
```

Such pairs of wait/signal are called **turnstiles**

- In general, reading a shared variable outside a lock may give an **inconsistent** value
- In this case, however, **only after the last thread** has arrived can any thread read  $nDone == n$ , because  $nDone$  is only incremented

# Reusable barriers

```
interface Barrier {  
    // block until expect() threads have reached barrier  
    void wait();  
  
    // number of threads expected at the barrier  
    int expect();  
}
```

Returned from

Reusable barrier: implement Barrier such that:

- a thread blocks on `wait()` until all threads have reached the barrier
- after `expect()` threads have executed `wait()`, the barrier is closed again

## Threads at a reusable barrier

Threads **continuously approach the barrier**, and all synchronize their access at the barrier

```
Barrier barrier = new Barrier(n); // barrier for n threads
```

---

thread<sub>k</sub>

```
while (true) {  
    // code before barrier  
    barrier.wait(); // synchronize at barrier  
    // code after barrier  
}
```

# Reusable barriers: first attempt

```
public class NonBarrier1 implements Barrier {
    int nDone = 0; // number of done threads
    Semaphore open = new Semaphore(0);
    final int n;

    // initialize barrier for `n' threads
    NonBarrier1(int n) {
        this.n = n;
    }

    // number of threads expected at the barrier
    int expect() {
        return n;
    }

    public void wait() {
        synchronized(this) {
            nDone += 1; // I'm done
            if (nDone == n)
                open.up(); // I'm the last arrived: All can go!
            open.down();
            open.up(); // proceed when possible
            // let the next one go
            synchronized(this) {
                nDone -= 1; // I've gone through
                if (nDone == 0)
                    open.down(); // I'm the last through: Close barrier!
            }
        }
    }
}
```

What if n threads “wait” here until  $nDone == n$ ?

More than one thread may open the barrier (the first `open.up()`): this was not a problem in the non-reusable version, but now some threads may be executing `wait` again before the barrier is **closed again!**

What if n threads “wait” here until  $nDone == 0$ ?

More than one thread may try to close the barrier (last `open.down()`): **Deadlock!**

# Reusable barriers: second attempt

```
public class NonBarrier2 implements Barrier {
    int nDone = 0; // number of done threads
    Semaphore open = new Semaphore(0);
    final int n;

    // initialize barrier for `n' threads
    NonBarrier2(int n) {
        this.n = n;
    }

    // number of threads expected at the barrier
    int expect() {
        return n;
    }

    public void wait() {
        synchronized(this) {
            nDone += 1;
            if (nDone == n) open.up();
        }
        open.down()
        open.up()

        synchronized(this) {
            nDone -= 1;
            if (nDone == 0) open.down();
        }
    }
}
```

```
// I'm done
// open barrier
// proceed when possible
// let the next one go
// I've gone through
// close barrier
```

Is multiple signalling possible? No!

Anything else going wrong?

A fast thread may **race through** the whole method, and re-enter it before the barrier has been closed, thus **getting ahead** of the slower threads (still in the previous iteration of the barrier)

This is not prevented by strong semaphores: it occurs because the last thread through leaves the gate open (calls `open.up()`)

# Reusable barriers: second attempt (cont'd)

```
1 public class NonBarrier2 {
2     public void wait() {
3         synchronized(this) {
4             nDone += 1;
5             if (nDone == n) open.up();
6             open.down();
7             open.up();
8             synchronized(this) {
9                 nDone -= 1;
10                if (nDone == 0) open.down();
11            }
12        }
13    }
14}
```

- (a) All  $n$  threads are at 8, with `open.count() == 1`
- (b) The fastest thread  $t_f$  completes `wait` and re-enters it with  $nDone = n - 1$
- (c) Thread  $t_f$  reaches 6 with  $nDone = n$ , which it can execute because `open.count() > 0`
- (d) Thread  $t_f$  reaches 8 again, but it is one iteration ahead of all other threads!

# Reusable barriers: Correct solution



Photo by Photnart: Heidelberg Lock, Germany

# Reusable barriers: Correct solution

```

public class SemaphoreBarrier implements Barrier {
    int nDone = 0; // number of done threads
    Semaphore gate1 = new Semaphore(0); // first gate
    Semaphore gate2 = new Semaphore(1); // second gate
    final int n;

    // initialize barrier for 'n' threads
    SemaphoreBarrier(int n) {
        this.n = n;
    }

    // number of threads expected at the barrier
    int expect() {
        return n;
    }

    public void wait() { approach(); leave(); }
  
```

gate1 closed  
gate2 open

```

    void approach() {
        synchronized (this) {
            nDone += 1; // arrived
            if (nDone == n) { // if last in:
                gate1.up(); // open gate1
                gate2.down(); // close gate2
            }
        }
        gate1.down(); // pass gate1
        gate1.up(); // let next pass
    }

    void leave() {
        synchronized (this) {
            nDone -= 1; // going out
            if (nDone == 0) { // if last out:
                gate2.up(); // open gate2
                gate1.down(); // close gate1
            }
        }
        gate2.down(); // pass gate2
        gate2.up(); // let next pass
    }
  
```

# Reusable barriers: improved solution

If the semaphores support **adding  $n$  to the counter** at once, we can write a barrier with fewer semaphore accesses

```

public class NSemaphoreBarrier extends SemaphoreBarrier {
    Semaphore gate1 = new Semaphore(0) ★ // first gate
    Semaphore gate2 = new Semaphore(0) ★ // second gate

    void approach() {
        synchronized (this) {
            nDone += 1;
            if (nDone == n)
                gate1.up(n);
            gate1.down(); // pass gate1
            // last thread here closes gate1
        }
    }

    void leave() {
        synchronized (this) {
            nDone -= 1;
            if (nDone == 0)
                gate2.up(n);
            gate2.down();
            // last thread here closes gate2
        }
    }
}
  
```

Both gates initially closed

Open gate1 for n threads

Open gate2 for n threads

Java semaphores support adding  $n$  to counter (`release(n)`)

Anyway, `up(n)` need not be uninterruptible, so we can also implement it with a loop

# Readers-writers

## Readers-writers: overview

Readers and writers concurrently access shared data:

- readers may execute concurrently with other readers, but need to exclude writers
- writers need to exclude both readers and other writers

The problem captures situations common in databases, filesystems, and other situations where accesses to shared data may be inconsistent



# Readers-writers: The problem

```
interface Board<T> {  
    // write message `msg' to board  
    void write(T msg);  
    // read current message on board  
    T read();  
}
```

**Readers-writers** problem: implement **Board** data structure such that:

- multiple reader can operate concurrently
- each writer has exclusive access

**Invariant:**  $\#Writers = 0 \vee (\#Writers = 1 \wedge \#Readers = 0)$

Other properties that a good solution should have:

- support an arbitrary number of readers and writers
- no starvation of readers or writers

# Readers and writers

Readers and writers continuously and asynchronously try to access the board, which must guarantee proper synchronization

Board<Message> board;

---

reader<sub>n</sub>

```
while (true) {  
    // read message from board  
    Message msg = board.read();  
    // do something with 'msg'  
    process(msg);  
}
```

writer<sub>m</sub>

```
while (true) {  
    // create a new message  
    Message msg = create();  
    // write 'msg' to board  
    board.write(msg);  
}
```

# Readers-writers board: write

```

public class SyncBoard<T> implements Board<T> {
    int nReaders = 0; // # readers on board
    Lock lock = new Lock(); // for exclusive access to nReaders
    Semaphore empty = new Semaphore(1); // 1 iff no active threads
    T message; // current message

    public T read() {
        lock.lock(); // lock to update nReaders
        if (nReaders == 0) // if first reader,
            empty.down(); // set not empty
        nReaders += 1; // update active readers
        lock.unlock(); // release lock to nReaders

        T msg = message; // read (critical section)

        lock.lock(); // lock to update nReaders
        nReaders -= 1; // update active readers
        if (nReaders == 0) // if last reader
            empty.up(); // set empty
        lock.unlock(); // release lock to nReaders
        return msg;
    }
}
  
```

Solution based on  
one lock and one  
semaphore

```

public void write(T msg) {
    // get exclusive access
    empty.down();
    message = msg; // write (cs)
    // release board
    empty.up();
}
  
```

**invariant** {  $nReaders == 0 \Leftarrow empty.count() == 1$  }

count() becomes 1 after executing empty.up()  
and it happens that  $nReaders = 0$

# Properties of the readers-writers solution

We can check the following **properties** of the solution:

- `empty` is a binary semaphore
- when a writer is running, no reader can run
- one reader waiting for a writer to finish also locks out other readers
- a reader signals “empty” only when it is the last reader to leave the board
- deadlock is not possible (no circular waiting)

However, **writers can starve**: as long as readers come and go with at least one reader always active, writers are shut out of the board.

# Readers-writers board without starvation

```

public class FairBoard<T> extends SyncBoard<T> {
    // held by the next thread to go
    Semaphore baton = new Semaphore(1, true); // fair binary sem.

    public T read() {
        // wait for my turn
        baton.down();
        // release a waiting thread
        baton.up();
        // read() as in SyncBoard
        return super.read();
    }

    public void write(T msg) {
        // wait for my turn
        baton.down();
        // write() as in SyncBoard
        super.write(msg);
        // release a waiting thread
        baton.up();
    }
}

```

One additional semaphore

## Readers-writers board: write

```

public class SyncBoard<T> implements Board<T> {
    int nReaders = 0; // # readers on board
    Lock lock = new Lock(); // for exclusive access to nReaders
    Semaphore empty = new Semaphore(1); // 1 iff no active threads
    T message; // current message

    public T read() {
        lock.lock(); // lock to update nReaders
        if (nReaders == 0) // if first reader,
            empty.down(); // set not empty
        nReaders += 1; // update active readers
        lock.unlock(); // release lock to nReaders

        T msg = message; // read (critical section)

        lock.lock(); // lock to update nReaders
        nReaders -= 1; // update active readers
        if (nReaders == 0) // if last reader
            empty.up(); // set empty
        lock.unlock(); // release lock to nReaders
        return msg;
    }

    public void write(T msg) {
        // get exclusive access
        empty.down();
        message = msg; // write (cs)
        // release board
        empty.up();
    }
}

invariant{nReaders == 0  $\Leftrightarrow$  empty.count() == 1}

```

If and only if

invariant breaks temporary here when  
 $nReaders = 0$ ; just before calling `empty.up()`

# Readers-writers board without starvation

```

public class FairBoard<T> extends SyncBoard<T> {
    // held by the next thread to go
    Semaphore baton = new Semaphore(1, true); // fair binary sem.

    public T read() {
        // wait for my turn
        baton.down();
        // release a waiting thread
        baton.up();
        // read() as in SyncBoard
        return super.read();
    }

    public void write(T msg) {
        // wait for my turn
        baton.down();
        // write() as in SyncBoard
        super.write(msg);
        // release a waiting thread
        baton.up();
    }
}
  
```

Now writers do not starve:

- Suppose a writer is waiting that all active readers leave: it waits on `empty.down()` while holding the baton
- If new readers arrive, they are shut out waiting for the baton
- As soon as the active readers terminate and leave, the writer is signaled `empty`, and thus it gets exclusive access to the board

## Readers-writers with priorities

The starvation free solution we have presented gives all threads the **same priority**: assuming a fair scheduler, writers and readers take turn as they try to access the board

In some applications it might be preferable to enforce **difference priorities**:

- $R = W$ : readers and writers have the same priority (as in FairBoard)
- $R > W$ : readers have higher priority than writers (as in SyncBoard)
- $W > R$ : writers have higher priority than readers

© 2016–2019 Carlo A. Furia, Sandro Stucki



Except where otherwise noted, this work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

To view a copy of this license, visit

<http://creativecommons.org/licenses/by-sa/4.0/>.

# Monitors

Lecture 5 of TDA384/DIT391

Principles of Concurrent Programming

Nir Piterman and Gerardo Schneider

Chalmers University of Technology | University of Gothenburg



UNIVERSITY OF  
GOTHENBURG



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY

# Today's menu

- Monitors
- Signaling disciplines
- Implementing monitors
- Monitors in Java
- Monitors: dos and don'ts

# Today's menu

- Monitors
  - Common patterns of synchronization
  - Language constructs solving synchronization
- Implementing monitors
  - Issues and problems
  - In depth understanding
  - Choice of right constructs
- Monitors in Java
  - Language constructs solving synchronization

## Learning outcomes

### *Knowledge and understanding:*

- demonstrate knowledge of the issues and problems that arise in writing correct concurrent programs;
- identify the problems of synchronization typical of concurrent programs, such as race conditions and mutual exclusion

### *Skills and abilities:*

- apply common patterns, such as lock, semaphores, and message-passing synchronization for solving concurrent program problems;
- apply practical knowledge of the programming constructs and techniques offered by modern concurrent programming languages;
- implement solutions using common patterns in modern programming languages

### *Judgment and approach:*

- evaluate the correctness, clarity, and efficiency of different solutions to concurrent programming problems;
- judge whether a program, a library, or a data structure is safe for usage in a concurrent setting;
- pick the right language constructs for solving synchronization and communication problems between computational units.

# Beyond semaphores

Semaphores provide a powerful, concise mechanism for synchronization and mutual exclusion

Unfortunately, they have several shortcomings:

- they are intrinsically global and unstructured: it is difficult to understand their behavior by looking at a single piece of code
  - they are prone to deadlocks or other incorrect behavior: it is easy to forget to add a single, crucial call to up or down
  - they do not support well different conditions
- 
- In summary semaphores are a low-level synchronization primitive
  - We will raise the level of abstraction

# Monitors

# Monitors

Monitors provide a **structured synchronization mechanism** built on top of object-oriented constructs – especially the notions of class, object, and encapsulation

In a **monitor class**:

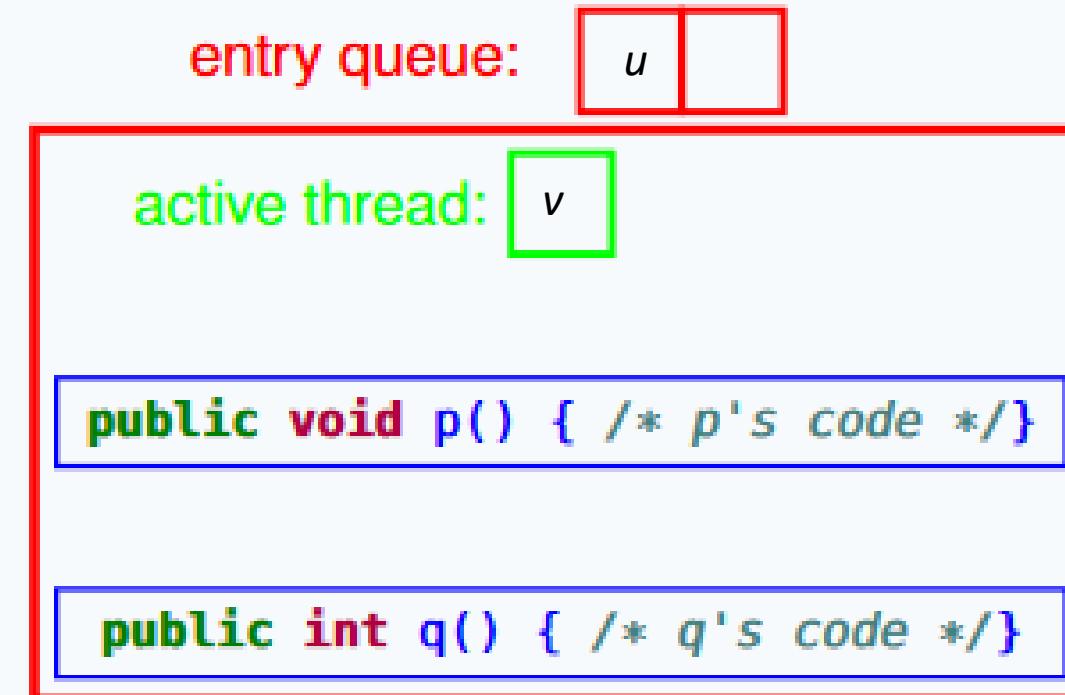
- attributes are **private**
- methods execute in **mutual exclusion**

A **monitor** is an object instantiating a monitor class that **encapsulates synchronization** mechanisms:

- **attributes** are shared variables, which all threads running on the monitor can see and modify
- **methods** define critical sections, with the built-in guarantee that at most one thread is active on a monitor at any time

## Monitors: entry queue

Threads trying to access a monitor **queue for entry**; as soon as the active thread leaves the monitor the next thread in the entry queue gets exclusive access to the monitor



## Monitors in pseudo-code

We declare monitor classes by adding the pseudo-code keyword `monitor` to regular Java classes

Note that `monitor` is **not** a valid Java keyword – that is why we highlight it in a different color – but we will use it to simplify the presentation of monitors

- Turning a pseudo-code monitor class into a proper Java class is straightforward:
  - mark all attributes as `private`
  - add `locking` to all public methods

Details on how to implement monitors in Java are presented later

Reminder: We also annotate monitor classes with `invariants` using the pseudo-code keyword `invariant`: **not** a valid Java keyword

# Counter monitor

A shared counter that is free from race conditions:

```
monitor class Counter {  
    int count = 0; // attribute, implicitly private  
  
    public void increment() { // method, implicitly atomic  
        count = count + 1;  
    }  
  
    public void decrement() { // method, implicitly atomic  
        count = count - 1;  
    }  
}
```

The implementation of monitors guarantees that multiple threads executing increment and decrement run in mutual exclusion

# Mutual exclusion for $n$ threads

Mutual exclusion for  $n$  threads accessing their critical sections is straightforward to achieve using monitors: every monitor method executes **uninterruptibly** because at most one thread is running on a monitor at any time

- A proper monitor implementation also guarantees **starvation freedom**

```

monitor class CriticalSection {
    T1 a1; T2 a2; ... // shared data

    public void critical1() {
        //  $t_1$ 's critical section
    }
    // ...
    public void criticaln() {
        //  $t_n$ 's critical section
    }
}

CriticalSection cs;


---


thread  $t_k$ 
while (true) {
    cs.criticalk();
    // non-critical section
}

```

# Condition variables

For synchronization patterns more complex than mutual exclusion, monitors provide **condition variables**

A **condition variable** is an instance of a class with interface:

```
interface Condition {  
    void wait();          // block until signal  
    void signal();        // signal to unblock  
    boolean isEmpty();   // is no thread waiting on this condition?  
}
```

A monitor class can declare condition variables as **attributes** (private, thus only callable by methods of the monitor)

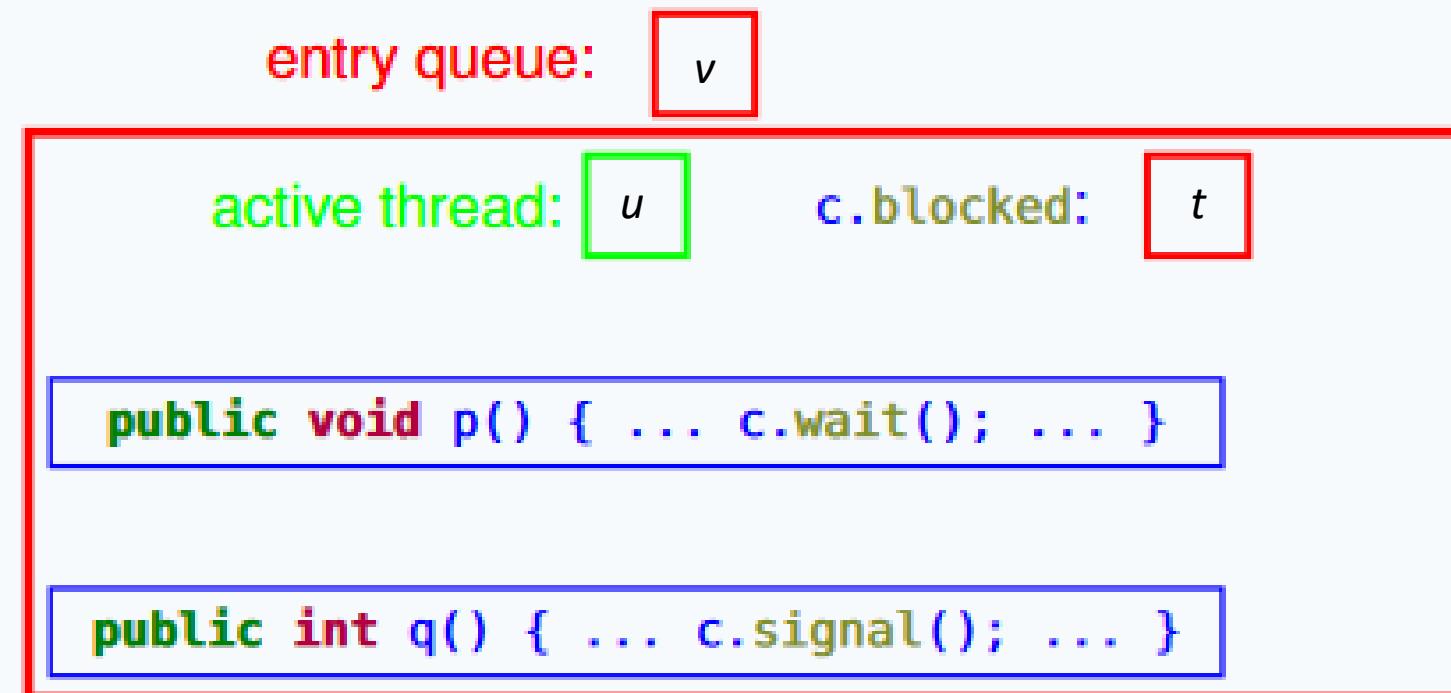
Every condition variable `c` includes a **FIFO queue** `blocked`:

- `c.wait()` blocks the running thread, appends it to `blocked`, and releases the lock on the monitor
- `c.signal()` removes one thread from `blocked` (if it's not empty) and unblocks it
- `c.isEmpty()` returns `true` iff `blocked` is empty

# Condition variables

Every condition variable `c` includes a **FIFO queue** blocked:

- `c.wait()` blocks the running thread, appends it to `blocked`, and releases the lock on the monitor
- `c.signal()` removes one thread from `blocked` (if it's not empty) and unblocks it



# Producer-consumer problem: recap

```
interface Buffer<T> {  
    // add item to buffer; block if full  
    void put(T item);  
  
    // remove item from buffer; block if empty  
    T get();  
  
    // number of items in buffer  
    int count();  
}
```

Producer-consumer problem: implement **Buffer** such that:

- producers and consumers access the buffer in mutual exclusion
- consumers block when the buffer is empty
- producers block when the buffer is full (bounded buffer variant)

# Producer-consumer with monitors: unbounded buffer

An implementation of **producer-consumer** with an **unbounded buffer** using **monitors**.

```

monitor class MonitorBuffer<T> implements Buffer<T> {
    Collection storage = ...; // any collection (list, set, ...)
    Condition notEmpty = new Condition(); // signal when not empty

    public void put(T item) {           No effect if there are no waiting consumers
        storage.add(item)             // store item
        notEmpty.signal();           // signal buffer not empty
    }

    public T get() {
        if (storage.count() == 0)    Get in queue waiting for an item
            notEmpty.wait();         // wait until buffer not empty
        return storage.remove();    // retrieve item
    }

    invariant { #storage.add == #notEmpty.signal }
  }

```

Assumption:  
 Exactly one thread wakes up.  
 No other changes to the state of the monitor.

Number of added  
 elements to buffer  
 equals number of  
 signaling

# Producer-consumer with monitors: bounded buffer

Producer-consumer with a **bounded** buffer (capacity is the maximum size)  
 uses two condition variables

```

monitor class BoundedMonitorBuffer<T> extends MonitorBuffer<T> {
    Condition notFull = new Condition(); // signal when not full

    public void put(T item) {
        if (storage.count() == capacity)
            notFull.wait();           // wait until buffer not full
        super.put(item);          // do as in MonitorBuffer.put(item)
    }

    public T get() {
        T item = super.get();      // do as in MonitorBuffer.get()
        notFull.signal();          // signal buffer not full
        return item;
    }
}
  
```

No other changes to the state of the monitor.  
 Assumption:  
 Exactly one thread woken up.

# Signaling disciplines

# Signaling disciplines

When a thread  $s$  calls `signal()` on a condition variable, it is executing inside the monitor

Since no more than one thread may be active on a monitor at any time, the thread  $u$  unblocked by  $s$  cannot enter the monitor immediately

The **signaling discipline** determines what happens to a signaling thread  $s$  after it unblocks another thread  $u$  by signaling

Two main choices of **signaling discipline**:

**signal and continue:**  $s$  continues executing;

$u$  is moved to the entry queue of the monitor

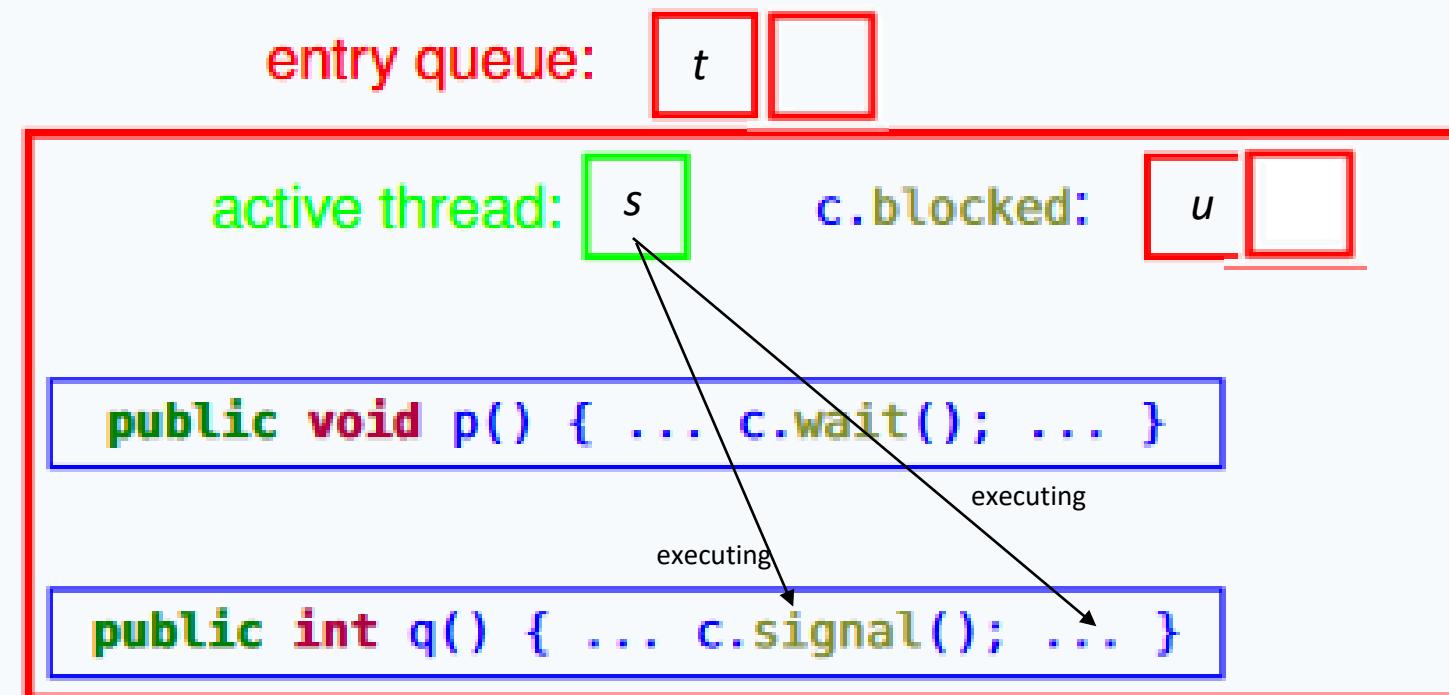
**signal and wait:**  $s$  is moved to the entry queue of the monitor

$u$  resumes executing (it silently gets the monitor's lock)

# Signal and continue

Under the **signal and continue** discipline:

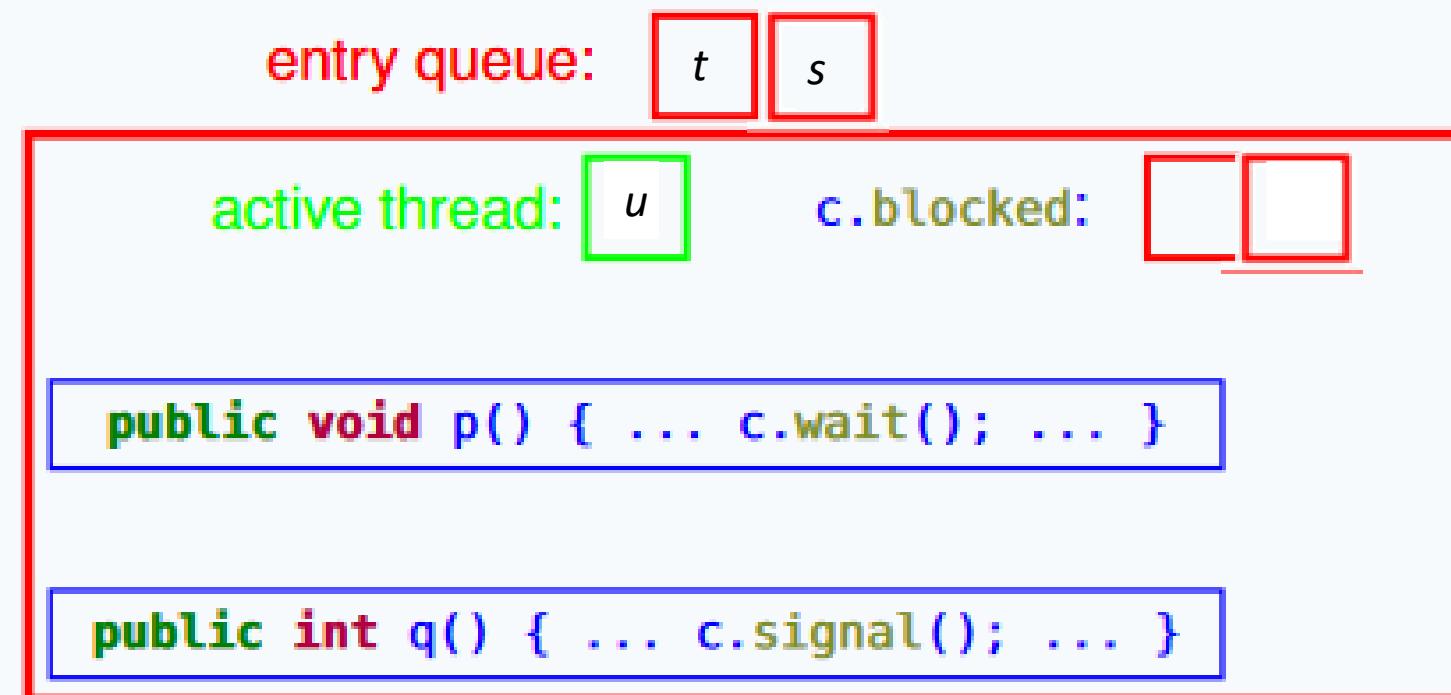
- the unblocked thread *u* is moved to the monitor's **entry queue**
- the signaling thread *s* **continues** executing



# Signal and wait

Under the **signal and wait** discipline:

- the signaling thread *s* is moved to the monitor's **entry queue**
- the unblocked thread *u* **resumes** executing



# Condition checking under different signaling disciplines

Under the **signal and wait** discipline, it is guaranteed that the **signaled condition holds** when the unblocked thread resumes execution – because it immediately follows the signal

In contrast, under the **signal and continue** discipline, the **signaled condition may no longer hold** when the unblocked thread ***u*** resumes execution – because the signaling thread, or other threads, may change the state while continuing

- Correspondingly, there are different patterns for **waiting on a condition variable** signaled as  
***if* (!buffer.isEmpty()) isNotEmpty.signal();**

## Signal and wait:

```
// check once
if (buffer.isEmpty())
    isNotEmpty.wait();
// here !buffer.isEmpty()
```

## Signal and continue:

```
// recheck after waiting
while (buffer.isEmpty())
    isNotEmpty.wait();
// here !buffer.isEmpty()
```

## Signal all

The **signal and continue** discipline does not guarantee that a thread resuming execution after a `wait` will find that the condition it has been waiting for is true: the signal is only a “hint”

- In spite of this shortcoming, most (if not all) implementations of monitors follow the **signal and continue** discipline – mainly because it is simpler to implement

Monitors following **signal and continue** typically also offer a condition-variable method:

```
void signalAll(); // unblock all threads blocked on this condition
```

This tends to be inefficient, because many threads will wake up only to discover the condition they have been waiting for is still not true, but works correctly with the waiting pattern using a loop (which is still not as inefficient as busy waiting!)

## More signaling disciplines

The **signaling discipline** determines what happens to a signaling thread *s* after it unblocks another thread *u* by signaling

Two variants of signal and continue and signal and wait are also sometimes used:  
**urgent signal and continue:** *s* continues executing;

*u* is moved to **the front of** the entry queue of the monitor

**signal and urgent wait:** *s* is moved to **the front of** the entry queue of the monitor;  
*u* resumes executing

To be precise:

- An urgent thread gets ahead of “regular” threads, but may have to queue behind other urgent threads that are waiting for entry
- This is implemented by adding a `urgentEntry` queue to the monitor, which has priority over the “regular” entry queue

# Signaling disciplines: Summary

A signaling discipline defines what happens to three **sets of threads**:

**S**: signaling threads

**U**: unblocked threads

**E**: threads in the entry queue

Write  $X > Y$  to denote that threads in set  $X$  have priority over threads in set  $Y$

- Then, different signaling policies can be expressed as:

**signal and continue**       $S > U = E$

**urgent signal and continue**       $S > U > E$

**signal and wait**       $U > S = E$

**signal and urgent wait**       $U > S > E$

Other combinations are also possible, but most of them do not make much sense in practice

# Implementing monitors

# Monitors from semaphores

We give an overview of how to **implement monitors using semaphores**

- This also rigorously defines the semantics of monitors:
  - Every monitor class uses a **strong semaphore entry** to model the entry queue
  - Every monitor method acquires **entry** upon entry and releases it upon exit

```
monitor class Counter {
    int x = 0;
  public void inc() {
    x = x + 1;
  }
}
```

```
class Counter {
    // strong/fair semaphore, initially 1
    Semaphore entry = new Semaphore(1, true);
    private int x = 0;
  public void inc() {
    entry.down();
    x = x + 1;
    entry.up();
  }
}
```

# Condition variables: Waiting

Every condition variable uses a queue blocked of threads waiting on the condition

```
abstract class WaitVariable implements Condition {  
    Queue<Thread> blocked = new Queue<Thread>(); // queue of blocked threads  
  
    // block until signal  
    public void wait() {  
        entry.up(); // release monitor lock  
        blocked.add(running); // enqueue running thread  
        running.state = BLOCKED; // set state as blocked  
    }  
  
    // is no thread waiting?  
    public boolean isEmpty() { return blocked.isEmpty(); }  
}
```

Reference to running thread

# Condition variables: Signal and continue

```
class SCVariable extends WaitVariable {  
    // signal to unblock  
    public void signal() {  
        if (!blocked.isEmpty()) {  
            Thread u = blocked.remove(); // u is the unblocked thread  
            entry.blocked.add(u); // u gets moved to entry queue  
            // the running, signaling thread continues executing  
        }  
    }  
}
```

The thread signaling continues its execution



# Condition variables: Signal and wait

```
class SWVariable extends WaitVariable {
    // signal to unblock
    public void signal() {
        if (!blocked.isEmpty()) {
            entry.blocked.add(running);      // the running, signaling thread
                                              // gets moved to entry queue
            Thread u = blocked.remove();    // u is the unblocked thread
            u.state = READY;               // set state as ready to run
            running.state = BLOCKED;       // set state as blocked
            // the unblocked, signaled thread resumes executing
        }
    }
}
```

# Semaphores from monitors

```
monitor class StrongSemaphore implements Semaphore {  
    int count;  
    Condition isPositive = new Condition(); // is count > 0?  
  
    public void down() {  
        if (count > 0)  
            count = count - 1;  
        else isPositive.wait();  
    }  
  
    public void up() {  
        if (isPositive.isEmpty())  
            count = count + 1;  
        else isPositive.signal();  
    }  
}
```

Can we implement semaphores using monitors?

Each signal matches a wait;  
thus no decrement or increment  
in the `else` branches

# Semaphores from monitors: A theoretical result

The result that monitors can implement semaphores (and vice versa) is important **theoretically: no expressiveness loss**

However, implementing a lower-level mechanism (semaphores) using a higher-level one (monitors) is **impractical** because it is likely to be inefficient

- If you have monitors use it (do not implement semaphores)

As usual, if you need monitors or semaphores use the efficient library implementations available in your programming language of choice

- Do not reinvent the wheel!

# Monitors in Java

## Two kinds of Java monitors

Java does not include full-fledged monitor classes, but it offers **support** to implement monitor classes following some **programming patterns**

There are two sets of **monitor-like primitives** in Java:

- **language based**: has been included since early versions of the Java language
- **library based**: has been included since Java 1.5

We have seen bits and pieces of both already, since they feature in simpler synchronization primitives as well

# Language-based monitors

A class JM can implement a monitor class M as follows:

- every **attribute** in JM is **private**
- every **method** in JM is **synchronized** – which guarantees it executes atomically

```
monitor class M {
    int x, y;

    public void p()
    { /* ... */ }

    public int q()
    { /* ... */ }
}
```

```
class JM {
    private int x, y;

    public synchronized void p()
    { /* ... */ }

    public synchronized int q()
    { /* ... */ }
}
```

This mechanism does **not guarantee fairness** of the entry queue associated with the monitor:  
 entry may behave like a **set**

# Language-based condition variables

Each language-based monitor implicitly include a single condition variable with signal and continue discipline:

- calling `wait()` **blocks** the running thread, waiting for a signal
- calling `notify()` **unblocks** any one thread waiting in the monitor
- calling `notifyAll()` **unblocks all** the threads waiting in the monitor

```
monitor class M {
    int x; Condition isPos;
    public void p()
    { while (x < 0)
        isPos.wait(); }
    public int q()
    { if (x > 0)
        isPos.signal(); }
}
```

```
class JM {
    private int x;
    public synchronized void p()
    { while (x < 0)
        wait(); }
    public synchronized int q()
    { if (x > 0)
        notify(); }
}
```

It does **not guarantee fairness** of the blocked threads queue: `blocked` may behave like a set

# How to wait in a language-based monitor

Calls to `wait()` always must be **inside a loop** checking a condition

- There are **multiple reasons** to do this:

- Under the signal and continue discipline, the signaled condition may be no longer true when an unblocked thread can run
- Since the blocked queue is not fair, the signaled condition may be “**stolen**” by a thread that has been waiting for less time
- Since there is a single implicit condition variable, the signal may represent a condition other than the one the unblocked thread is waiting for
- In Java (and other languages), spurious wakeups are possible: a waiting thread may be unblocked even if no thread signaled.

A class `LM` can implement a monitor class `M` using **explicit locks**:

- add a private `monitor` attribute – a **fair lock**
- every **method** in `CM` starts by locking `monitor` and ends by unlocking `monitor` – which guarantees it executes atomically

```
monitor class M
{
    int x, y;
    public void p()
    { /* ... */ }

    class LM {
        private final Lock monitor = new ReentrantLock(true); // fair lock
        private int x, y;

        public void p()
        {
            monitor.lock();
            /* ... */
            monitor.unlock();
        }
    }
}
```

This mechanism **guarantees fairness** of the entry queue associated with the monitor: blocked behaves like a **queue**

# Library-based condition variables

Condition variables with signal and continue discipline can be generated by a monitor's lock:

```
monitor class M {  
  
    Condition isXPos  
    = new Condition();  
    Condition isYPos  
    = new Condition();  
  
    int x, y;  
    // ...  
}
```

```
class JM {  
    private final Lock monitor  
    = new ReentrantLock(true);  
    private final Condition isXPos  
    = monitor.newCondition();  
    private final Condition isYPos  
    = monitor.newCondition();  
  
    private int x, y;  
    // ...  
}
```

## Library-based condition variables (cont'd)

Each library-based **condition variable**  $c$  has *signal and continue* discipline:

- calling  $c.\text{await}()$  **blocks** the running thread, waiting for a signal
  - calling  $c.\text{signal}()$  **unblocks** any one thread waiting on  $c$
  - calling  $c.\text{signalAll}()$  **unblocks all** the threads waiting on  $c$
- 
- When  $\text{signalAll}()$  is called, the ordering of lock reacquisition is also fair (same order as in  $\text{blocked}$ ) – provided the lock itself is fair
  - These methods must be called **while holding the lock** used to generate the condition variable; otherwise, an `IllegalMonitorStateException` is thrown

This mechanism **guarantees fairness** of the queue of blocked threads associated with the condition variable:  $\text{blocked}$  behaves like a queue

# How to wait in a library-based monitor

Calls to `await()` always must be **inside a loop** checking a condition

There are **multiple reasons** to do this (compare to the case of language-based monitors):

- Under the signal and continue discipline, the signaled condition may no longer be true when an unblocked thread can run
- In Java (and other languages), spurious wakeups are possible: a waiting thread may be unblocked even if no thread signaled

## Threads, interrupted

Waiting operations (in [monitors](#) as well as in [semaphores](#)) may be **interrupted** by some low-level code that calls a thread's `interrupt()` method

- This is apparent in the signature of the waiting methods, which typically may throw an object of type `InterruptedException`: interrupting a waiting thread wakes up the thread, which has to handle the exception
- We normally [ignore](#) the case of [interrupted threads](#), since it belongs to lower-level programming
  - When calling waiting primitives, you typically propagate the exception to the main method (or simply catch and ignore it)

## Threads, interrupted (cont'd)

It is important that programs ensure that an interrupted thread still leaves the system in a consistent state by **releasing all locks** it holds

- In **language-based monitors**, an interrupted thread in a **synchronized** method automatically releases the monitor's lock
- In library-based monitors, use a **finally** block to release the monitor's lock in case of exception:

```
class LM {  
    private final Lock monitor = new ReentrantLock(true);  
  
    public void p() {  
        monitor.lock();  
        try { /* ... */ }  
        finally { monitor.unlock(); }  
    }  
}
```

# Monitors: dos and don'ts

# Nested monitor calls

What happens if a method in monitor  $M$  calls a method  $n$  in monitor  $N$  (with condition variable  $c_N$ )? Different **rules** are possible:

1. Prohibit nested calls
  2. Release lock on  $M$  before acquiring lock on  $N$
  3. Hold lock on  $M$  while also locking  $N$ 
    - 3.1 When waiting on  $c_N$  release both locks on  $N$  and on  $M$
    - 3.2 When waiting on  $c_N$  release only lock on  $N$
- Rules 3 are prone to deadlock – especially rule 3.2. – because deadlocks often occur when trying to acquire multiple locks
  - Java monitors (both language- and library-based) follow the deadlock-prone rule 3.2
    - **Rule of thumb:** avoid nested monitor calls as much as possible
    - Note that if  $N$  is the same object as  $M$ , nested calls are not a problem (the implicit locks are reentrant)

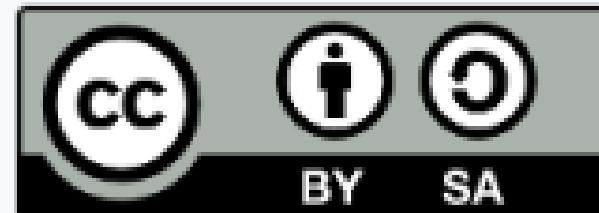
## Monitors: Pros

- Monitors provide a **structured** approach to concurrent programming, which builds atop the familiar notions of objects and encapsulation
- This **raises** the level of **abstraction** of concurrent programming compared to semaphores.
- Monitors introduce **separation of concerns** when programming concurrently:
  - mutual exclusion is implicit in the use of monitors,
  - condition variables provide a clear means of synchronization.

## Monitors: Cons

- Monitors generally have a larger **performance overhead** than semaphores
  - Performance must be traded against error proneness
- The different **signaling disciplines** are a source of confusion, which tarnishes the clarity of the monitor abstraction. In particular, signal and continue is both less intuitive (because a condition can change before a waiting thread has a chance to run on the monitor) and the most commonly implemented discipline
- For complex synchronization patterns, **nested monitor calls** are another source of complications

© 2016–2019 Carlo A. Furia, Sandro Stucki



Except where otherwise noted, this work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

To view a copy of this license, visit

<http://creativecommons.org/licenses/by-sa/4.0/>.

# Parallelizing computations

TDA384/DIT391

Principles of Concurrent Programming

Nir Piterman and Gerardo Schneider

Chalmers University of Technology | University of Gothenburg



UNIVERSITY OF  
GOTHENBURG



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY

# Lesson's menu

- Challenges to parallelization
- Fork/join parallelism
- Pools and work stealing

# Lesson's menu

- Challenges to parallelization
  - evaluate efficiency of different solution
- Fork/join parallelism
  - programming constructs
- Pools and work stealing
  - Programming constructs, efficiency

## Learning outcomes

### *Knowledge and understanding:*

- demonstrate knowledge of the issues and problems that arise in writing correct concurrent programs;
- identify the problems of synchronization typical of concurrent programs, such as race conditions and mutual exclusion

### *Skills and abilities:*

- apply common patterns, such as lock, semaphores, and message-passing synchronization for solving concurrent program problems;
- apply practical knowledge of the programming constructs and techniques offered by modern concurrent programming languages;
- implement solutions using common patterns in modern programming languages

### *Judgment and approach:*

- evaluate the correctness, clarity, and efficiency of different solutions to concurrent programming problems;
- judge whether a program, a library, or a data structure is safe for usage in a concurrent setting;
- pick the right language constructs for solving synchronization and communication problems between computational units.

# Parallelization: risks and opportunities

Concurrent programming introduces:

- + the potential for parallel execution (faster, better resource usage)
- the risk of race conditions (incorrect, unpredictable computations)

The main challenge of concurrent programming is thus introducing parallelism without affecting correctness

There is no panacea!

We show several (common?) paradigms where some difficulties can be mitigated.

# Paradigms of parallelization

In this lesson, we explore several **paradigms** to parallelizing computations in multi-processor systems

A **task**  $(F, D)$  consists in computing the result  $F(D)$  of applying function  $F$  to input data  $D$

A **parallelization** of  $(F, D)$  is a collection  $(F_1, D_1), (F_2, D_2), \dots$  of tasks such that  $F(D)$  equals the composition of  $F_1(D_1), F_2(D_2), \dots$

We discuss how to parallelize such problems in the context of **shared-memory models** (such as **Java threads**).

We note that similar solutions are possible in **Erlang** using **message-passing** between processes.

# Challenges to Parallelization

# Challenges to parallelization

A strategy to **parallelize** a task  $(F, D)$  should be:

- **correct**: the overall result of the parallelization is  $F(D)$
- **efficient**: the total resources (time and memory) used to compute the parallelization are less than those necessary to compute  $(F, D)$  sequentially

A number of factors **challenge** designing correct and efficient **parallelizations**:

- sequential dependencies
- synchronization costs
- spawning costs
- error proneness and composability

# Sequential dependencies

- Some steps in a task computation depend on the result of other steps; this creates **sequential dependencies** where one task must wait for another task to run
- Sequential dependencies **limit** the amount of parallelism that can be achieved

For example, to compute the sum  $1 + 2 + \dots + 8$  we could split into:

- a. computing  $1 + 2, 3 + 4, 5 + 6, 7 + 8$
- b. computing  $(1 + 2) + (3 + 4)$  and  $(5 + 6) + (7 + 8)$
- c. computing  $((1 + 2) + (3 + 4)) + ((5 + 6) + (7 + 8))$

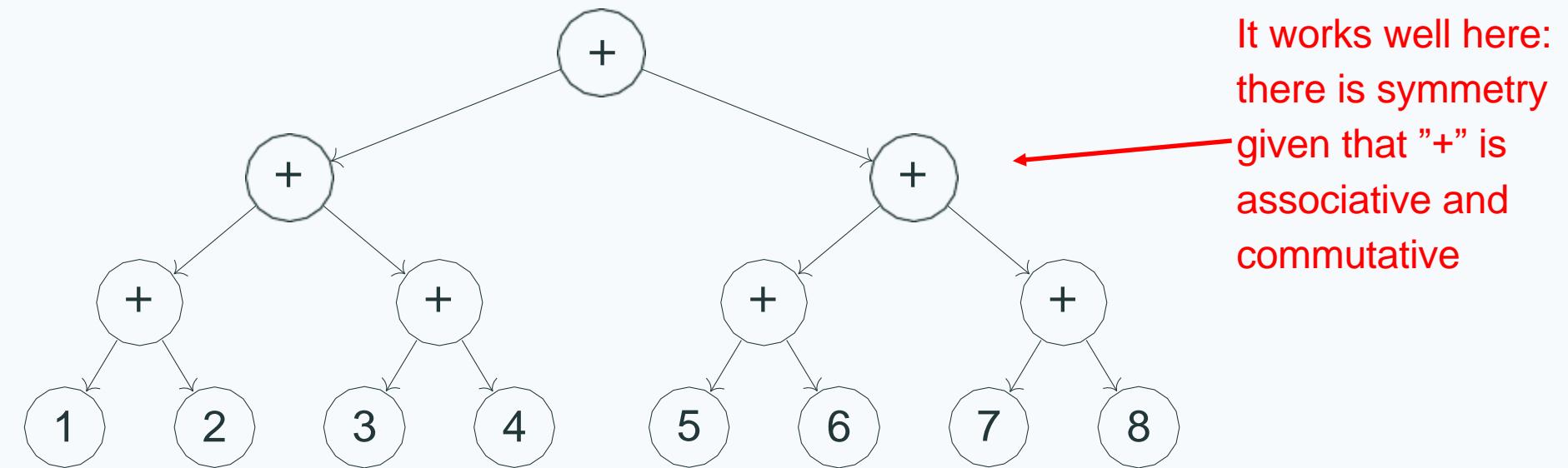
The computations in each group **depend** on the computations in the previous group, and hence the corresponding tasks must execute **after** the latter have completed

The **synchronization problems** (producer-consumer, dining philosophers, etc.) we discussed capture kinds of **sequential dependencies** that may occur when parallelizing

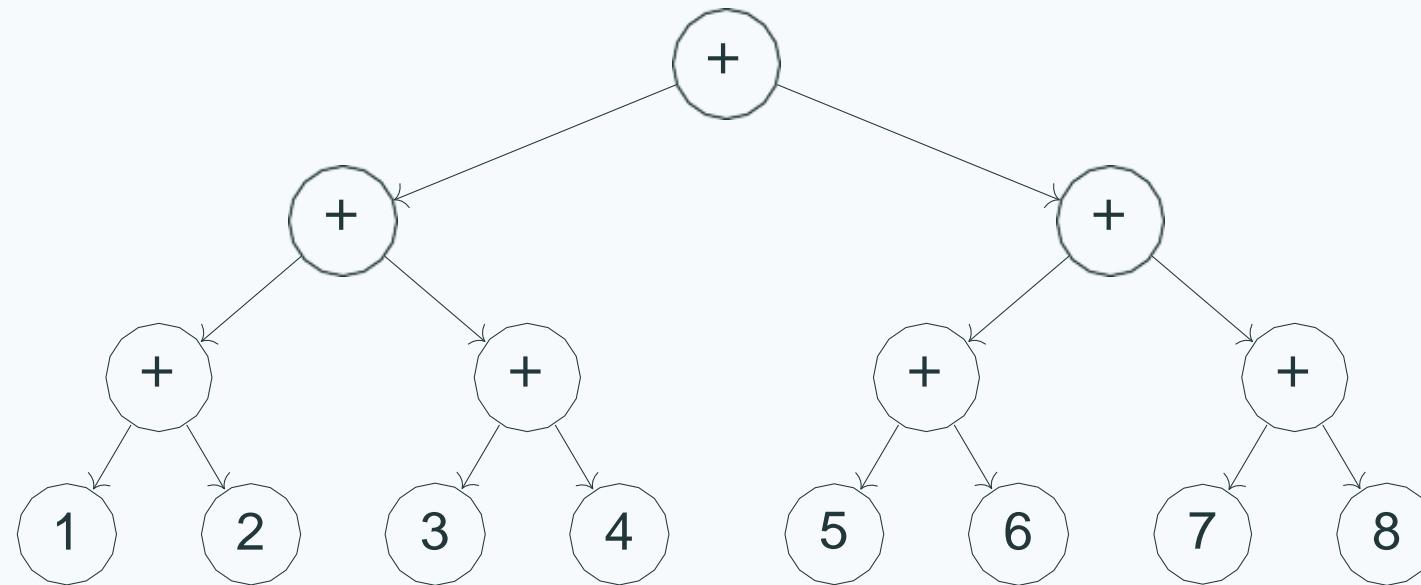
# Dependency graph

We represent tasks as the **nodes in a graph**, with arrows connecting a task to the ones it **depends** on

The graph must be **acyclic** for the decomposition to be executable



# Dependency graph



The time to compute a node is the **maximum** of the times to compute its children plus the time computing the node itself

Assuming all operations take a similar time, the **longest path** from the root to a leaf is proportional to the optimal running time with parallelization (ignoring overhead and assuming all processes can run in parallel)

# Digression: some latency numbers

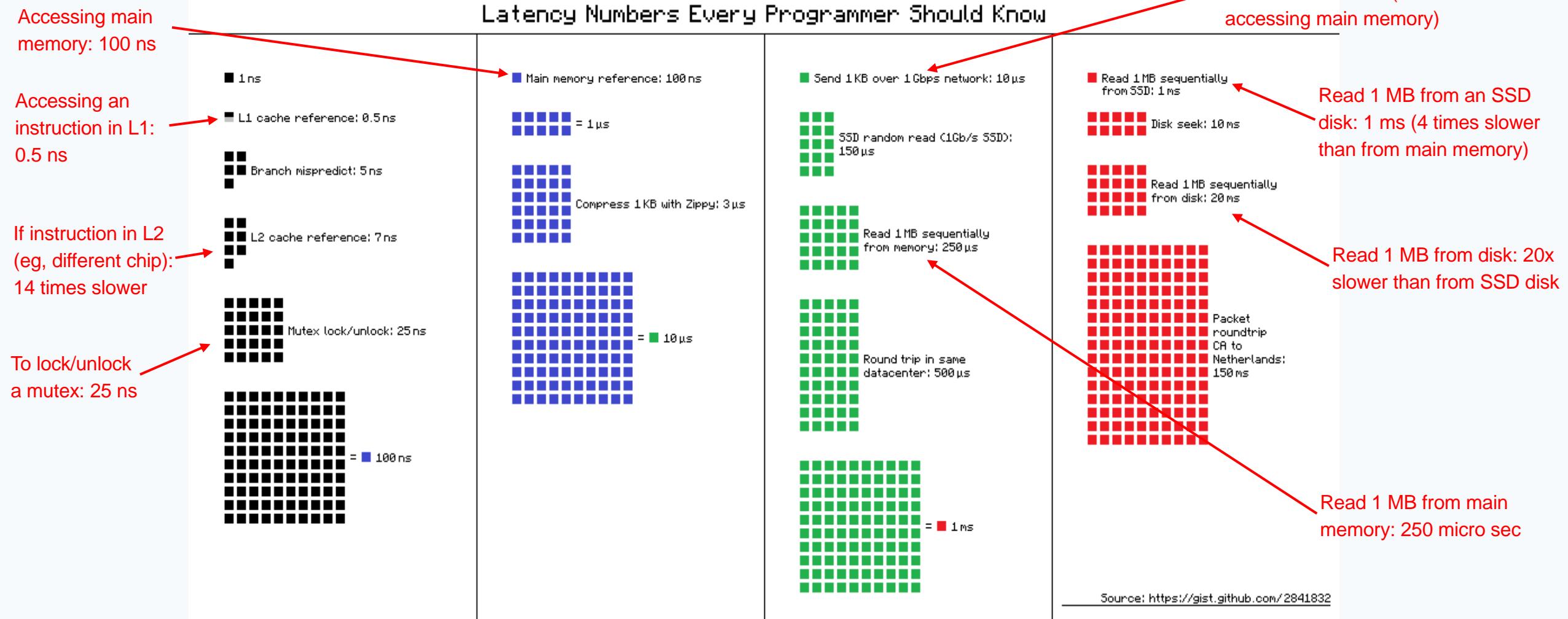


Chart by [ayshen](#), based on Peter Norvig's "[Teach Yourself Programming in Ten Years](#)"

More numbers at <https://gist.github.com/hellerbarde/2843375>

# Synchronization costs

Synchronization is required to preserve correctness, but it also introduces overhead that add to the overall cost of parallelization

In shared-memory concurrency:

- synchronization is based on locking
- locking synchronizes data from cache to main memory, which may involve a 100x overhead
- other costs associated with locking may include context switching (wait/signal) and system calls (mutual exclusion primitives)

In message-passing concurrency:

- synchronization is based on messages
- exchanging small messages is efficient, but sending around large data is quite expensive (still goes through main memory)
- other costs associated with message passing may include extra acknowledgment messages and mailbox management (removing unprocessed messages)

# Spawning costs

Creating a new process is generally **expensive** compared to sequential function calls within the same process, since it involves:

- reserving memory
- registering the new process with runtime system
- setting up the process's local memory (stack and mailbox)

Even if process creation is increasingly **optimized**, the cost of spawning should be **weighted against** the speed up that can be obtained by additional parallelism

In particular, when the processes become way more than the available processors, there will be diminishing returns with more spawning

# Error proneness and composability

Synchronization is **prone to errors such as data races, deadlocks, and starvation**

From the point of view of software construction, the lack of **composability** is a challenge that prevents us from developing parallelization strategies that are **generally applicable**

# Error proneness and composability

Consider an **Account** class with methods **deposit** and **withdraw** that execute **atomically**.  
What happens if we combine the two methods to implement a **transfer** operation?

```
class Account {  
    synchronized void deposit(int amount)  
    { balance += amount; }  
  
    synchronized void withdraw(int amount)  
    { balance -= amount; }  
}
```

execute uninterruptedly

```
class TransferAccount  
    extends Account {  
  
    // transfer from 'this' to 'other'  
    void transfer(int amount, Account other)  
    { this.withdraw(amount);  
        other.deposit(amount); }  
}
```

Method **transfer** does **not** execute **uninterruptedly**: other threads can execute between the call to **withdraw** and the call to **deposit**, possibly preventing the transfer from succeeding

(For example, **Account other** may be closed; or the total balance temporarily looks lower than it is!)

# Composability

```
class Account {
    void // thread unsafe!
    deposit(int amount)
    { balance += amount; }
    void // thread unsafe!
    withdraw(int amount)
    { balance -= amount; }
}
```

```
class TransferAccount
    extends Account {
    // transfer from 'this' to 'other'
    synchronized void
    transfer(int amount, Account other)
    { this.withdraw(amount);
    other.deposit(amount); }
```

None of the **natural solutions to composing** is fully satisfactory:

- let clients of Account do the locking where needed – error proneness, revealing implementation details, scalability
- recursive locking – risk of deadlock, performance overhead

With **message passing**, we encounter similar problems – synchronizing the effects of messaging two independent processes

# Sequential dependencies and spawning costs

A number of factors **challenge** designing correct and efficient **parallelizations**:

- sequential dependencies
- synchronization costs
- spawning costs
- error proneness and composability

In the rest of **this lesson**, we present:

- **fork/join parallelism** – naturally capture sequential dependencies
- **pools** – curb spawning costs

In future lessons we will see other approaches to reduce synchronization costs and achieving composability

# Fork/join parallelism

# Mitigating Spawning Costs

In the rest of **this lesson**, we present:

- **fork/join parallelism** – naturally capture sequential dependencies
- **pools** – curb spawning costs

Apply to problems that look like this:

A **task**  $(F, D)$  consists in computing the result  $F(D)$  of applying function  $F$  to input data  $D$

A **parallelization** of  $(F, D)$  is a collection  $(F_1, D_1), (F_2, D_2), \dots$  of tasks such that  $F(D)$  equals the composition of  $F_1(D_1), F_2(D_2), \dots$

What kind of problems look like this?

# Recursion: merge sort

```
// Allocate space and call recursive merge
// sort.
static void mergeSort(int[] arr, int size) {
    int[] space = new int[size];
    mergeSortRec(arr, 0, size, space);
}

// Recursive merge sort
static void mergeSortRec(int[] arr,
                        int low,
                        int high,
                        int[] space) {
    if (high - low <= 1) return;
    int mid = low + (high - low) / 2;
    mergeSortRec(arr, low, mid, space);
    mergeSortRec(arr, mid, high, space);
    merge(arr, low, mid, high, space);
}
```

```
static void merge(int[] arr, int low,
                  int mid, int high,
                  int[] space) {
    int i = low; int j = mid; int k = low;
    while (i < mid && j < high)
    {
        if (arr[i] <= arr[j])
            space[k++] = arr[i++];
        else
            space[k++] = arr[j++];
    }
    while (i < mid)
        space[k++] = arr[i++];
    while (j < high)
        space[k++] = arr[j++];
    for (i = low; i < high; i++)
        arr[i] = space[i];
}
```

# Parallel recursion

```
// Allocate space and call recursive merge
// sort.
static void mergeSort(int[] arr, int size) {
    int[] space = new int[size];
    mergeSortRec(arr, 0, size, space);
}

// Recursive merge sort
static void mergeSortRec(int[] arr,
                        int low,
                        int high,
                        int[] space) {
    if (high - low <= 1) return;
    int mid = low + (high - low) / 2;
    mergeSortRec(arr, low, mid, space);
    mergeSortRec(arr, mid, high, space);
    merge(arr, low, mid, high, space);
}
```

```
// This function calls recursive merge sort.
public static void mergeSort(int[] arr, int size)
{
    int[] space = new int[size];
    MergeSortParallel m = new
        MergeSortParallel(arr, 0, size, space);
    m.run();
    mergeSortRec(arr, 0, size, space);
}

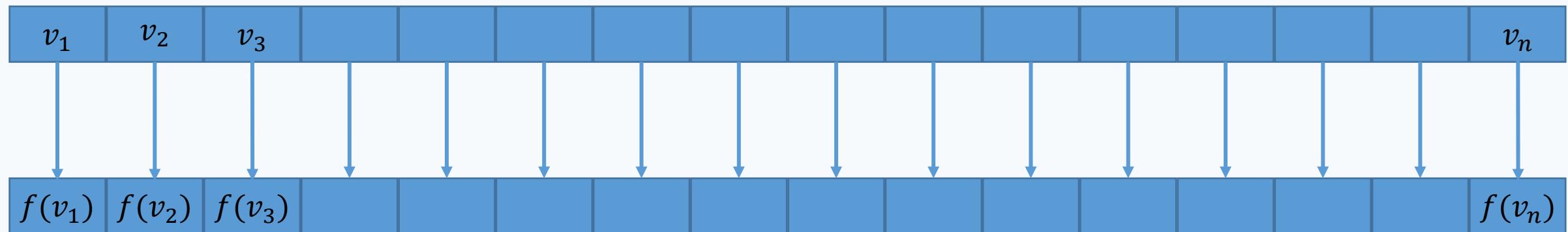
static class MergeSortParallel extends Thread {
    public void run() {
        if (high - low <= 1) return;
        int mid = low + (high - low) / 2;
        Thread l = new
            MergeSortParallel(arr, low, mid, space);
        Thread r = new
            MergeSortParallel(arr, mid, high, space);

        l.start(); r.start();

        try {
            l.join(); r.join();
            merge(arr, low, mid, high, space);
        } catch (InterruptedException e) {}
    }
}
```

# Map / forEach

- Apply a given function to all elements in a collection.
- Natural concept in functional programming.
- Introduced also in imperative programming languages (C++, Java, ...).
- The approach to doing this in Java here is structured towards concurrency ...



# Parallel map

The lack of interference in `map` lends itself to parallelization

```
public class Map<X,Y> {

    protected ArrayList<X> source;
    protected Function<X,Y> func;
    protected ArrayList<Y> target;

    Map(ArrayList<X> source,
        Function<X,Y> func,
        ArrayList<Y> target) {
        ...
    }

    public void map() {
        for (int i=0 ; i< source.size(); i++) {
            target.set(i,func.apply(source.get(i)));
        }
    }
}
```

```
public class ParallelMap<X,Y> extends Map<X,Y> {

    ParallelMap(ArrayList<X> source, ... ) { ... }

    public class Applicator implements Runnable {
        int loc;
        Applicator(int loc) { ... }

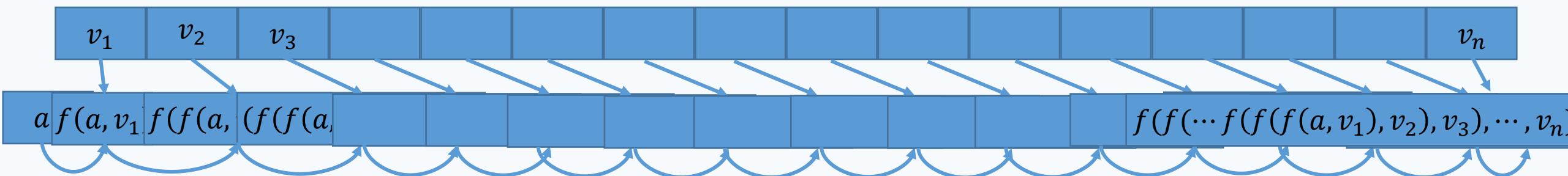
        public void run() {
            target.set(loc,func.apply(source.get(loc)));
        }
    }

    public void map() {
        ArrayList<Thread> threads =
            new ArrayList<Thread>(source.size());

        for (int i=0 ; i<source.size() ; i++) {
            threads.set(i,new Thread(new Applicator(i)));
            threads.get(i).start();
        }
        try {
            for (int i=0 ; i<source.size() ; i++) {
                threads.get(i).join();
            }
        } catch (InterruptedException e) {}
    }
}
```

# Reduce – summarize a collection

- Apply a given function starting from an initial value and accumulating the result applied to all elements in a collection.
- Natural concept in functional programming.
- Introduced also in imperative programming languages (C++, Java, ...).
- The approach to doing this in Java here is structured towards concurrency ...



# Parallel reduce

The parallel version of `reduce` (aka `foldr`) uses a halving strategy similar to merge sort

```
import java.util.function.BinaryOperator;  
  
public class Reduce<X> {  
  
    protected ArrayList<X> source;  
    protected BinaryOperator<X> func;  
    protected X initial;
```

Parallel reduce equals reduce if:

- Function  $F$  is associative (parallel reduce does not apply  $F$  right-to-left)
- For every list element  $E$ :

$$F(E, \text{init}) = F(\text{init}, E) = E$$

(The data is a monoid with  $F$  as the binary operation and  $\text{init}$  its identity element)

It works with e.g. addition but not division

```
import java.util.function.BinaryOperator;  
  
public class ParallelReduce<X> extends Reduce<X> {  
  
    ParallelReduce(...) { ... }  
  
    public class Applicator extends Thread {  
        Applicator(int st, int end, X init) { ... }  
  
        public void run() {  
            if (end - st > 1) {  
                int mid = st + (end - st) / 2;  
                Thread l = new Applicator(st, mid, init);  
                Thread r = new Applicator(mid, end, init);  
                l.start(); r.start();  
                try {  
                    l.join(); r.join();  
                    source.set(st, func.apply(  
                        source.get(st), source.get(mid)));  
                } catch (InterruptedException e) {}  
            } else {  
                source.set(st, func.apply(initial,  
                    source.get(st)));  
            }  
        }  
    }  
    reduce() {  
        Applicator a = new Applicator(0, size, initial);  
        a.run();  
        return source.get(0);  
    }  
}
```

# MapReduce

**MapReduce** is a **programming model** based on parallel distributed variants of the primitive operations `map` and `reduce`

MapReduce is a somewhat more general model, since it may produce a list of values from a list of key/value pairs, but the underlying ideas are the same

MapReduce implementations typically work on **very large, highly parallel, distributed databases or filesystems.**

- The original MapReduce implementation was proprietary developed at Google
- **Apache Hadoop** offers a widely-used open-source Java implementation of MapReduce

# Revisiting parallel merge sort

There are a number of things that should be improved in the parallel merge sort example:

```
granularity too small!  
  
protected void run() {  
    if (high - low <= 1) return;          // size <= 1: sorted already  
    int mid = low + (high - low)/2;      // mid point  
    // left and right halves:  
    PMergeSort left = new PMergeSort(data, low, mid);  
    PMergeSort right = new PMergeSort(data, mid, high);  
    left.fork();                         // fork thread working on left  
    right.fork();                        // fork thread working on right  
    left.join();                         // wait for sorted left half  
    right.join();                        // wait for sorted right half  
    merge(mid);                         // merge halves  
}
```

the forking thread is idle!

# Revisited parallel merge sort using fork/join

```
protected void run() {  
    if (high - low <= THRESHOLD)  
        sequential_sort(data, low, high);  
  
    else {  
        int mid = low + (high - low)/2;           // mid point  
        // left and right halves  
        PMergeSort left = new PMergeSort(data, low, mid);  
        PMergeSort right = new PMergeSort(data, mid, high);  
        left.fork();      // fork thread working on left  
        right.run();     // continue work on right  
        left.join();     // when done, wait for sorted left half  
        merge(mid);      // merge halves  
    }  
}
```

choose experimentally (at least 1000)

before joining, do more work in current task

# Fork/join good practices

In order to obtain **good performance** using fork/join parallelism:

- After forking children tasks, keep some **work for the parent** task before it joins the children
- For the same reason, use `invoke` and `invokeAll` **only at the top** level as a norm
- Perform **small** enough **tasks sequentially** in the parent task, and fork children tasks only when there is a **substantial chunk** of work left
  - Java's fork/join framework recommends that each task be assigned between 100 and 10'000 basic computational steps
- Make sure different tasks can **proceed independently** – minimize data dependencies

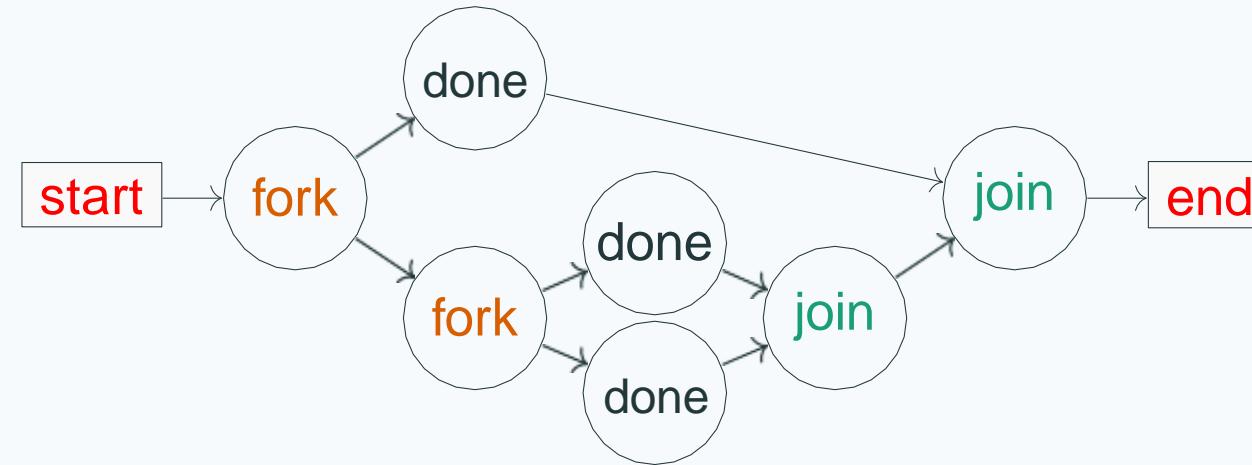
The advantages of parallelism may only be visible with several **physical processors**, and on very **large inputs**

(The Java runtime may need to warm up before it optimizes the parallel code more aggressively)

# Fork/join parallelism

This recursive subdivision of a task that assigns new processes to smaller tasks is called **fork/join parallelism**:

- **forking**: spawning child processes and assigning them smaller tasks
- **joining**: waiting for the child processes to complete and combining their results



The **order** in which we **wait** at a **join** node for forked children does not affect the total waiting time: if we wait for a slower process first, we won't wait for the others later

# What are the issues?

- The number of threads depends on the problem and may choke the computing power.
- How to return a value?

## Two similar solutions

- Fork/Join pools
  - Recursively partition the task – what to do with fork and join?
  - Granularity of tasks decreases
  - Load is hard to evenly distribute
  - Terminates
- Executor services
  - Get a bank of tasks and run them in parallel
  - Meant for larger and more predictable coarse-grained tasks
  - Dependencies are simpler
  - Can be terminated

# Pools and work stealing

# How many processes is *lagom*?

Parallelizing by following the recursive structure of a task is simple and appealing

However, the potential **performance gains** should be weighted against the **overhead** of creating and running many processes

- Process creation in **Erlang** is **lightweight**:  
1 GB of memory fits about 432'000 processes, so one million processes is quite feasible



# How many processes is lagom?

Parallelizing by following the recursive structure of a task is simple and appealing

However, the potential **performance gains** should be weighted against the **overhead** of creating and running many processes

- There are still limits to how many processes fit in memory
- Besides, even if we have enough memory, more processes don't improve performance if their number greatly exceeds the number of available physical processors



Remember Amdahl's law



# Workers and pools

**Process pools** are a technique to address the problem of using an **appropriate number of processes**

A pool creates a number of **worker** processes upon initialization

The number of workers is chosen according to the actual **available** resources to run them in parallel – a detail which pool users need **not** know about:

- As long as more work is available, the pool **deals** a work assignment to a worker that is available
- The pool **collects** the results of the workers' computations
- When all work is completed, the pool terminates and returns the overall **result**

This kind of pool is called a **dealing pool**: it actively deals work to workers

# Workers

**Workers** are threads that run as long as the pool that created them does

A **worker** can be in one of two states:

- **idle**: waiting for work assignments from the pool
- **busy**: computing a work assignment

```
public class WorkThread
{ Queue [] queue; // queues of all worker threads
public void run() {
{ int me = ThreadID.get(); // my thread id
while (true) {
    for (Task task: queue[me]) // run all tasks in my queue
        task.run();
    if (queue[me].empty()) queue[me].await();
} } }
```

# From dealing to stealing

**Dealing pools** work well if:

- the workload can be split in **even chunks**, and
- the workload does **not change** over time (for example if users send new tasks or cancel tasks dynamically)

Under these conditions, the workload is balanced evenly between workers, so as to maximize the amount of parallel computation

In **realistic applications**, however, these conditions are not met:

- it may be **hard to predict** reliably which tasks take more time to compute the workload is **highly dynamic**

**Stealing pools** use a different approach to allocating tasks to workers that better addresses these challenging conditions

# Work stealing

A **stealing pool** associates a **queue** to every worker process

The pool distributes new tasks by adding them to the workers' queues

When a worker becomes **idle**:

- first, it gets the next task from **its own queue**
- if its queue is empty, it can directly **steal** tasks from the queue of another worker that is currently busy

With this approach, workers adjust dynamically to the current working conditions without requiring a supervisor that can reliably predict the workload required by each task

With stealing, the pool may even send all tasks to **one default thread**, letting other idle threads steal directly from it, simplifying the pool and reducing the synchronization costs it incurs

# Work stealing algorithm

Outline of the algorithm  
for **work stealing**

It assumes the queue  
array queue can be  
accessed by concurrent  
threads without race  
conditions

```
public class WorkStealingThread
{ Queue [] queue; // queues of all worker threads
public void run()
{ int me = ThreadID.get(); // my thread id
while (true) {
    for (Task task: queue[me]) // run all tasks in my queue
        task.run();
    // now my queue is empty: select another random thread
    int victim = random.nextInt(queue.length);
    // try to take a task out of the victim's queue
    Task stolen = queue[victim].pop();
    // if the victim's queue was not empty, run the stolen task
    if (stolen != null) stolen.run();
} } }
```

# Fork/join

# Characteristics

- Dynamic forking and joining
- Granularity of tasks changing
- Load hard to even
- Termination of task

# Fork/join parallelism in Java

Java package **java.util.concurrent** includes a library for **fork/join** parallelism

To implement a method  $T\ m()$  using fork/join parallelism:

If  $m$  is a **procedure** ( $T$  is **void**):

- create a class that inherits from `RecursiveAction`
- override **void compute()** with  $m$ 's computation

If  $m$  is a **function**:

- create a class that inherits from `RecursiveTask<T>`
- override **T compute()** with  $m$ 's computation

`RecursiveAction` and `RecursiveTask<T>` provide methods:

- **fork()**: schedule for asynchronous parallel execution
- $T\ join()$ : waits for termination and returns result if  $T\ !=\ void$
- $T\ invoke()$ : arranges synchronous parallel execution (fork and join) and returns result if  $T\ !=\ void$
- **invokeAll(Collection<T> tasks)**: invoke all tasks in collection (fork all and join all), and return collection of results

# Parallel merge sort using fork/join

```

public class PMergeSort                                @Override
extends RecursiveAction {
  // values to be sorted:
  private Integer[] data;
  // to be sorted: data[low..high]:
  private int low, high;

  protected void compute() {
    if (high - low <= 1) return; // size<=1: sorted already
    int mid = low + (high - low)/2; // mid point
    // left and right halves:
    PMergeSort left = new PMergeSort(data, low, mid);
    PMergeSort right = new PMergeSort(data, mid, high);
    left.fork(); // fork thread working on left
    right.fork(); // fork thread working on right
    left.join(); // wait for sorted left half
    right.join(); // wait for sorted right half
    merge(mid); // merge halves
  }
}

```

# Running a fork/join task

The top computation of a fork/join task is started by a **pool** object:

// to sort array ‘numbers’ using PMergeSort:

```
RecursiveAction sorter = new PMergeSort(numbers, 0, numbers.length);
```

// schedule ‘sorter’ for execution, and wait for computation to finish:

```
ForkJoinPool.commonPool().invoke(sorter);
```

// now ‘numbers’ is sorted

The pool takes care of efficiently **dispatching work to threads**

The framework introduces a layer of **abstraction** between computational **tasks** and actual running **threads** that execute the tasks

This way, the fork/join model **simplifies** parallelizing computations, since we can focus on how to **split data** among tasks in a way that avoids race conditions

ForkJoinPool makes top invocation:

- it launches a pool object, a synchronous parallel execution of all threads which will fork and join
- it terminates once all the threads join and terminate



# Fork/join good practices

To take advantage of the number of available cores (in Java):

*“In Java, the fork/join framework provides support for parallel programming by splitting up a task into smaller tasks to process them using the available CPU cores.*

46/43

*When you execute ForkJoinPool() it creates an instance with a number of threads equal to the number returned by the method Runtime.getRuntime().availableProcessors(), using defaults for all the other parameters.”*

(Taken from <https://www.pluralsight.com/guides/introduction-to-the-fork-join-framework>)

# Executor Services

# Characteristics

- Large coarse-grained tasks
- Dependencies are simpler
- Meant to stay there until terminated

# Executing “things” in parallel in Java

Java package **java.util.concurrent** includes a library for **ExecutorServices**

Do you need to return a value?

If **m** is a **procedure**:

- Re-use the **Runnable** interface
- override **void run()** with **m**'s computation

If **m** is a **function**:

- Implement the **Callable<T>** interface
- override **T call()** with **m**'s computation
- allowed to throw!

External to **Runnable/Callable**:

- **Future<T>**: handle for waiting for termination, cancelling, returned results, and exception handling
- **fork()/join()**: are not over-ridden!
- **submit()/execute()**: using an appropriate service
- **invokeAll(Collection<Y extends Callable<T>> tasks)**: invoke all tasks in collection and wait for them to terminate

# Executor Services – implementing Thread pools

Java offers efficient implementations of **thread pools** in package **java.util.concurrent**

The **interface ExecutorService** provides:

- Schedule thread for execution: **void execute(Runnable thread)**:
- Schedule thread for execution, and return a Future object (to cancel the execution, or wait for termination): **Future submit(Runnable thread)**  
**Future<T> submit(Callable<T> call)**

Implementations of **ExecutorService** with different characteristics can also be obtained by factory methods of **class Executors**:

- **CachedThreadPool**: thread pool of dynamically variable size
- **workStealingPool**: thread pool using work stealing
- **ForkJoinPool**: work-stealing pool for running fork/join tasks – **careful w details!**
- ...

# Thread pools in Java: example

Without thread pools:

```
Counter counter = new Counter();
// threads t and u

Thread t = new Thread(counter);
Thread u = new Thread(counter);
t.start(); // increment once
u.start(); // increment twice
try { // wait for termination
  t.join(); u.join();
}
catch (InterruptedException e)
{
  System.out.println("Int!");
}
```

With thread pools:

```
Counter counter = new Counter();
// threads t and u

Thread t = new Thread(counter);
Thread u = new Thread(counter);
ExecutorService pool = Executors.newWorkStealingPool();
// schedule t and u for execution
Future<?> ft = pool.submit(t);
Future<?> fu = pool.submit(u);
try {
  ft.get(); fu.get();
}
catch (InterruptedException | ExecutionException e){
  System.out.println("Int!");
}
```

we use "?" since we are not interested in the result but use the future just for the sake of cancelling the task

# Parallel map vs executor map

```
import java.util.function.Function;

public class ParallelMap<X,Y> extends Map<X,Y> {

    ParallelMap(X[] source, ... ) ...

    public class Applicator implements Runnable {
        ...
    }

    public void map() {
        Thread[] threads = new Thread[size];

        for (int i=0 ; i<size ; i++) {
            threads[i] = new Thread(new Applicator(i));
            threads[i].start();
        }
        try {
            for (int i=0 ; i<size ; i++) {
                threads[i].join();
            }
        } catch (InterruptedException e) {}
    }
}
```

How would you implement it?

# Parallel map vs executor map

```

import java.util.function.Function;

public class ParallelMap<X,Y> extends Map<X,Y> {

    ParallelMap(X[] source, ... ) ...

    public class Applicator implements Runnable {
        ...
    }

    public void map() {
        Thread[] threads = new Thread[size];

        for (int i=0 ; i<size ; i++) {
            threads[i] = new Thread(new Applicator(i));
            threads[i].start();
        }
        try {
            for (int i=0 ; i<size ; i++) {
                threads[i].join();
            }
        } catch (InterruptedException e) {}
    }
}
  
```

```

public class ParallelMapPool<X,Y> extends Map<X,Y> {

    ParallelMapPool(ArrayList<X> source, ... ) ...

    public class Applicator implements Runnable {
        ...
    }

    public void map() {
        ExecutorService pool = Executors.newCachedThreadPool();

        for (int i=0 ; i<source.size() ; i++) {
            pool.execute(new Applicator(i));
        }
        pool.shutdown();
        try {
            pool.awaitTermination(1, TimeUnit.DAYS);
        }
        catch (InterruptedException e) {
        }
    }
}
  
```

# More executor maps

```
public class Applicator1 implements Callable<Y> {
    int loc;
    Applicator1(int loc) {
        this.loc = loc;
    }

    public Y call() {
        return func.apply(source.get(loc));
    }
}
```

```
public void map1() {
    ExecutorService pool = Executors.newCachedThreadPool();
    ArrayList<Future<Y>> futures = new
                                ArrayList<Future<Y>>(source.size());
    for (int i=0 ; i<source.size() ; i++) {
        futures.set(i,pool.submit(new Applicator1(i)));
    }

    for (int i=0 ; i<target.size() ; i++) {
        try {
            target.set(i,futures.get(i).get());
        }
        catch (InterruptedException e) {
            // Here we end up if this
            // thread was interrupted

            // You might want to wait again
        }
        catch (ExecutionException e) {
            // Here we end up if the
            // execution of the thread had an exception

            // You might want to run it again
        }
    }
    pool.shutdownNow();
}
```

# More executor maps

```
public class Applicator2 implements Callable<Y> {  
    X val;  
    Applicator2(X val) {  
        this.val = val;  
    }  
  
    public Y call() {  
        return func.apply(this.val);  
    }  
}
```

```
public void map2() {  
    ExecutorService pool = Executors.newCachedThreadPool();  
    ArrayList<Future<Y>> futures = new  
    ArrayList<Future<Y>>(source.size());  
    for (int i=0 ; i<source.size() ; i++) {  
        futures.set(i,pool.submit(new Applicator2(source.get(i))));  
    }  
  
    for (int i=0 ; i<source.size() ; i++) {  
        try {  
            target.set(i,futures.get(i).get());  
        }  
        catch (InterruptedException e) {  
            // Here we end up if this  
            // thread was interrupted  
  
            // You might want to wait again  
        }  
        catch (ExecutionException e) {  
            // Here we end up if the  
            // execution of the thread had an exception  
  
            // You might want to run it again  
        }  
    }  
    pool.shutdownNow();  
}
```

# More about the Future ...

*“A **Future** represents the result of an asynchronous computation.*

*Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation.*

*The result can only be retrieved using method **get** when the computation has completed, blocking if necessary until it is ready.*

*Cancellation is performed by the **cancel** method.*

*Additional methods are provided to determine if the task completed normally or was cancelled.*

*Once a computation has completed, the computation cannot be cancelled.*

*If you would like to use a Future for the sake of cancellability but not provide a usable result, you can declare types of the form **Future<?>** and return null as a result of the underlying task.”*

From the Java documentation about “public interface Future<V>”

# Process pools in Erlang

Erlang provides some load distribution services in the system module `pool`

These are aimed at distributing the load between different **nodes**, each a full-fledged collection of processes

© 2016–2019 Carlo A. Furia, Sandro Stucki



Except where otherwise noted, this work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

To view a copy of this license, visit

<http://creativecommons.org/licenses/by-sa/4.0/>.

# Parallel map with workers

We can define a parallel version of `map` using a pool:

map: apply function F to  
all elements in list L  
(independently)

```
pmap(F, L, N) -> init_pool( F, % function to be mapped
                                L, % workload: list to be mapped
                                fun ([H|T]) -> {H,T} end, % split: take first element
                                fun (R,Res) -> [R|Res] end, % join: cons with list
                                [], % initial value
                                N   % number of workers
).
```

Note that the `order` of the results may change from run to run

It is possible to restore the original order by using a more complex join function

# Functional Programming and Erlang

Lecture 6 of TDA384/DIT391

Principles of Concurrent Programming



UNIVERSITY OF  
GOTHENBURG

Nir Piterman and Gerardo Schneider

Chalmers University of Technology | University of Gothenburg

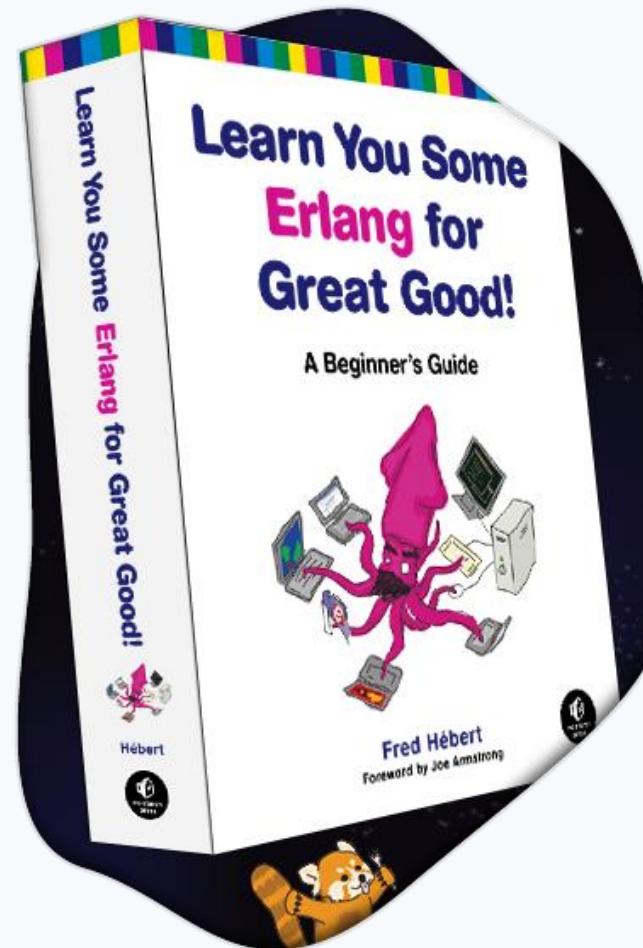


**CHALMERS**  
UNIVERSITY OF TECHNOLOGY

# Today's menu

- What is Erlang?
- Types
- Expressions and patterns
- Function definitions
- Recursion
- Impure and higher-order functions

Don't forget



(<http://learnyousomeerlang.com/>)

# What is Erlang?

# What is Erlang?

Erlang combines a functional language with message-passing features:

- The functional part is sequential, and is used to define the behavior of processes
- The message-passing part is highly concurrent: it implements the actor model, where actors are Erlang processes

This lecture covers the functional/sequential part of Erlang

# Erlang: A minimal history

**1973** *Hewitt and others develop the actor model* – a formal model of concurrent computation

**1985** *Agha further refines the actor model*

**Mid 1980s** *Armstrong and others at Ericsson prototype the first version of Erlang (based on the actor model)*

**Late 1980s** Erlang's implementation becomes efficient; Erlang code is used in production at Ericsson

**1998** Ericsson bans Erlang, which becomes open-source

**Late 2000s** Erlang and the actor model make a come-back in mainstream programming



# Erlang in the real world

Erlang has made a significant **impact** in the **practice** of concurrent programming by making the formal actor model applicable to real-world scenarios

- Initially, Erlang was mainly used for **telecommunication software**:
  - Ericsson's AXD301 switch – includes over one million lines of Erlang code; achieves “nine 9s” availability (99.999999%)
  - Cellular communication infrastructure (services such as SMSs)
- Recently, it has been rediscovered for Internet **communication apps**:
  - WhatsApp’s communication services are written in Erlang
  - Facebook Chat (in the past)

# Why Erlang?

*We've faced many challenges in meeting the ever-growing demand for [the WhatsApp] messaging services, but [...] Erlang continues to prove its capability as a versatile, reliable, high-performance platform.*

*Rick Reed, 2014 - [That's 'Billion' with a 'B': Scaling to the next level at WhatsApp](#)*

*The language itself has many pros and cons, but we chose Erlang to power [Facebook] Chat because its model lends itself well to concurrent, distributed, and robust programming.*

*Chris Piro, 2010 – [Chat Stability and Scalability](#)*

# What is a functional language?

**Functional** languages are based on elements quite **different from** those **imperative** languages are based on

**Imperative languages** (such as Java)  
are based on:

- state – variables
- state modifications - assignments
- iteration – loops

**Functional languages** (such as Erlang)  
are based on:

- data – values
- functions on data – without side effects
- functional forms – function composition, higher-order functions

# What is a functional language?

Functional languages are based on elements quite **different from** those imperative languages are based on

Imperative languages (such as Java)  
are based on:

An imperative program is the sequence of state modifications on variables

```
// compute xn
int power(int x, int n) {
    int result = 1;
    for (int i = n; i < n; i++)
        result *= x;
    return result;
}
```

Functional languages (such as Erlang)  
are based on:

A functional program is the side-effect-free application of functions on values

```
% compute XN
power(X, 0) -> 1;
power(X, N) -> X * power(X, N-1)
```

In functional programs, variables store **immutable** values, which can be **copied** but **not modified**

# The Erlang shell

You can experiment with Erlang using its [shell](#), which can evaluate expressions on the fly without need to define complete programs

```
$ erl
Erlang R16B03 (erts-5.10.4) [source] [64-bit] [smp:2:2]

Eshell V5.10.4 (abort with ^G)

1> 1 + 2.                      % evaluate expression `1 + 2'
3
2> c(power).                   % compile file `power.erl'
{ok,power}
3> power:power(2, 3).          % evaluate power(2, 3)
8
```

- Notice you have to terminate all expressions with a period
- Functions are normally defined in external files, and then used in the shell
- Compilation targets bytecode by default

# Types

# Types, dynamically

A **type** constrains:

1. The (kinds) of **values** that an expression can take
2. The **functions** that can be applied to expressions of that type

For example, the **integer** type:

1. includes integer values (1, -100, 234), but not, say, decimal numbers (10.3, -4.3311) or strings ("hello!", "why not")
2. supports functions such as sum +, but not, say, logical **and**

- Erlang is **dynamically typed**:

- programs do **not** use **type declarations**
- the type of an expression is only determined **at runtime**
  - when the expression is evaluated
- if there is a type mismatch (for example 3+false) expression evaluation **fails**
- Erlang types include **primitive** and **compound** data types

# An overview of Erlang types

Erlang offers eight **primitive types**:

- **Integers**: arbitrary-size integers with the usual operations
- **Atoms**: roughly corresponding to identifiers
- **FLOATS**: 64-bit floating point numbers
- **References**: globally unique symbols
- **Binaries**: sequences of bytes
- **Pids**: process identifiers
- **Ports**: for communication
- **Funs**: function closures

And three + two **compound types** (a.k.a. **type constructors**):

- **Tuples**: fixed-size containers
- **Lists**: dynamically-sized containers
- **Maps**: key-value associative tables (a.k.a. dictionaries) –recent feature, experimental in Erlang/OTP R17
- **Strings**: syntactic sugar for sequences of characters
- **Records**: syntactic sugar to access tuple elements by name

# Numbers

Numeric types include **integers** and **floats**

- We will mainly use *integers*, which are arbitrary-size, and thus do not overflow

EXPRESSION	VALUE	
3	3	explicit constant (“term”)
1 + 3	4	addition
1 - 3	-2	subtraction
4 * 2	8	multiplication
5 <b>div</b> 4	1	integer division
5 <b>rem</b> 3	2	integer remainder
5 / 4	1.25	float division
<b>power</b> (10,1000)	100000000...	no overflow!
2#101	5	101 in base 2
16#A1	161	A1 in base 16

# Atoms

Atoms are used to denote distinguished values

(they are similar to symbolic uninterpreted constants)

An atom can be:

- A sequence of alphanumeric characters and underscores, starting with a lowercase letter
- An arbitrary sequence of characters (including spaces and escape sequences) between single quotes
  - An atom is to be enclosed in single quotes (' ) if it does not begin with a lower-case letter or if it contains other characters than alphanumeric characters, underscore (\_), or @

Examples of valid atoms:

```
x
a_Longer_Atom
'Uppercase_Ok_in_quotes'
'This is crazy!'
true
```

# Booleans

In Erlang there is **no Boolean type**

Instead, the **atoms** `true` and `false` are conventionally used to represent Boolean values

OPERATOR	MEANING
<code>not</code>	negation
<code>and</code>	conjunction (evaluates both arguments/eager)
<code>or</code>	disjunction (evaluates both arguments/eager)
<code>xor</code>	exclusive or (evaluates both arguments/eager)
<code>andalso</code>	conjunction (short-circuited/lazy)
<code>orelse</code>	disjunction (short-circuited/lazy)

Examples:

```
true or      (10 + false)    % error: type mismatch in second argument
true orelse  (10 + false)    % true: only evaluates first argument
```

# Relational operators

Erlang's **relational operators** have a few syntactic differences with those of most other programming languages

OPERATOR	MEANING
<	less than
>	greater than
=<	less than or equal to
=>=	greater than or equal to
=:=	equal to
=/=	not equal to
==	numeric equal to
/=	numeric not equal to

Examples:

```
3 =:= 3      % true: same value, same type
3 =:= 3.0    % false: same value, different type
3 == 3.0    % true: same value, type not checked
```

# Order between different types

Erlang defines an **order relationship** between values of **any type**

When different types are compared, the following **order** applies:

*number < atom < reference < fun < port < pid < tuple < map < list*

Thus, the following inequalities hold:

3 < true  
3 < false  
999999999 < infinity  
100000000000000 < epsilon

⋮ number < atom  
⋮ number < atom  
⋮ number < atom  
⋮ number < atom

When comparing **tuples to tuples**:

- comparison is by size first
- two tuples with the same size or two lists are compared element by element, and satisfy the comparison only if all (existing) pairs satisfy it

# Tuples

Tuples denote **ordered sequences** with a **fixed** (but arbitrary for each tuple instance) **number of elements** (They are written as comma-separated sequences enclosed in **curly braces**)

Examples of valid tuples:

```
{ }                                % empty tuple
{ 10, 12, 98 }
{ 8.88, false, aToM }
{ 10, { -1, true } }
```

% elements may have different types  
% tuples can be nested

Functions on a tuple T:

Examples:

<code>element(2, {a, b, c})</code>	% b: tuples are numbered from 1
<code>setelement(1, {a, b}, z)</code>	% {z, b}
<code>tuple_size({ })</code>	% 0

FUNCTION	RETURNED VALUE
<code>element(N, T)</code>	Nth element of T
<code>setelement(N, T, X)</code>	a copy of T, with the Nth element replaced by X
<code>tuple_size(T)</code>	number of elements in T

# Lists

**Lists** denote **ordered sequences** with a **variable** (but immutable for any list instance) **number of elements** (They are written as comma-separated sequences enclosed in **square brackets**)

Examples of valid lists:

```
[ ]                                % empty list
[ 10, 12, 98 ]
[ 8.88, false, {1, 2} ]            % elements may have different type
[ 10, [ -1, true ] ]              % lists can be nested
```

# List operators

Some useful functions on lists  $L$ :

FUNCTION	RETURNED VALUE
<code>length(L)</code>	number of elements in $L$
<code>[H   L]</code>	a copy of $L$ with $H$ added as first (“head”) element
<code>hd(L)</code>	$L$ ’s first element (the “head”)
<code>tl(L)</code>	a copy of $L$ without the first element (the “tail”)
<code>L1 ++ L2</code>	the concatenation of lists $L1$ and $L2$
<code>L1 -- L2</code>	a copy of $L1$ with all elements in $L2$ removed (without repetitions, and in the order they appear in $L1$ )

Operator `|` is also called `cons`; using it, we can define any list:

```
[1, 2, 3, 4] =:= [1 | [2 | [3 | [4 | []]]]]
hd([H | T]) =:= H
tl([H | T]) =:= T
% this is an example of --
[1, 2, 3, 4, 2] -- [1, 5, 2] =:= [3, 4, 2]
```

# Strings

Strings are sequences of characters enclosed between double quotation marks

- Strings are just *syntactic sugar* for lists of character codes

String concatenation is implicit whenever multiple strings are juxtaposed without any operators in the middle

Using strings ( $\$c$  denotes the integer code of character  $c$ ):

```
""                      % empty string =:= empty list
"hello!"
"hello" "world"      % =:= "helloworld"
"xyz" =:= [$x, $y, $z] =:= [120, 121, 122] % true
[97, 98, 99]          % evaluates to "abc"!
```

# Records

Records are ordered sequences with a fixed number of elements, where each element has an atom as name

- Records are just *syntactic sugar* for tuples where positions are named

```
% define `person' record type
% with two fields: `name' with default value "add name"
%                   `age' without default value (undefined)
-record(person, { name="add name", age })
% `person' record value with given name and age
#person{name="Joe", age=55}
#person{age=35, name="Jane"} % fields can be given in any order
% when a field is not initialized, the default applies
#person{age=22} =:= #person{name="add name", age=22}
% evaluates to `age' of `Student' (of record type `person')
Student#person.age
```

- Erlang's shell does not know about records, which can only be used in modules
  - In the shell: #person{age=7, name="x"} is {person, "x", 7}.

# Expressions and patterns

# Variables

Variables are identifiers that can be bound to values

(they are similar to constants in an imperative programming language)

A variable name is a sequence of alphanumeric characters, underscores, and @, starting with an uppercase letter or an underscore

In the shell, you can directly bind values to variable:

- Evaluating `Var = expr` binds the value of expression `expr` to variable `Var`, and returns such value as value of the whole binding expression
- Each variable can only be bound once
- To clear the binding of variable `Var` evaluate `f(Var)`
- Evaluating `f()` clears all variable bindings
- The anonymous variable `_` (“any”) is used like a variable whose value can be ignored

In modules, variables are used with pattern matching (which we present later)

# Expressions and evaluation

- **Expressions** are evaluated exhaustively to a **value** – sometimes called (ground) **term**: a number, an atom, a list, ...

The **order of evaluation** is given by the usual **precedence rules**

(using **parentheses** forces the evaluation order to be inside-out of the nesting structure)

Some **precedence rules** to be aware of:

- **and** has higher precedence than **or**
- **andalso** has higher precedence than **orelse**
- when lazy (**andalso**, **orelse**) and eager (**and**, **or**) Boolean operators are mixed, they all have the same precedence and are left-associative
- **++** and **--** are right-associative (concatenation and subtraction in lists)
- relational operators have lower precedence than Boolean operators; thus you have to use parentheses in expressions such as `(3 > 0) and (2 == 2.0)`

# Precedence rules: Examples

<code>3 + 2 * 4</code>	⇒ is 11
<code>3 + (2 * 4)</code>	⇒ is 11
<code>(3 + 2) * 4</code>	⇒ is 20
true <b>or</b> false <b>and</b> false	⇒ is true
true <b>orelse</b> false <b>andalso</b> false	⇒ is true
true <b>or</b> false <b>andalso</b> false	⇒ is false
true <b>orelse</b> false <b>and</b> false	⇒ is true (why?)

After evaluating the first “true”  
 there is no need to evaluate the rest



# Patterns

Pattern matching is a flexible and concise mechanism to bind values to variables

It is widely used in functional programming languages to define functions on data (especially lists); Erlang is no exception

A pattern has the same structure as a term, but in a pattern some parts of the term may be replaced by free variables

Examples of patterns:

3  
A  
{X, Y}  
{X, 3}  
[H | T]  
[H | [2]]

- Note that a pattern may contain bound variables
  - in this case, evaluating the pattern implicitly evaluates its bound variables

# Pattern matching

**Pattern matching** is the process that, given a pattern  $P$  and a term  $T$ , binds the variables in  $P$  to match the values in  $T$  according to  $P$  and  $T$ 's structure

If  $P$ 's structure (or type) cannot match  $T$ 's, pattern matching **fails**

PATTERN = TERM	BINDINGS
$3 = 3$	none
$A = 3$	$A: 3$
$A = B$	if $B$ is bound then $A := B$ ; otherwise fail
$\{X, Y\} = 3$	fail (structure mismatch)
$\{X, Y\} = \{1, 2\}$	$X: 1, Y: 2$
$\{X, Y\} = \{"a", [2, 3]\}$	$X: "a", Y: [2, 3]$
$[H T] = [1, 2]$	$H: 1, T: [2]$
$[H [2]] = [1, 2]$	$H: 1$
$[F, S] = [foo, bar]$	$F: foo, S: bar$
$\{X, Y\} = [1, 2]$	fail (type mismatch)

# Pattern matching: Notation

Given a **pattern**  $P$  and a **term**  $T$ , we write  $\langle P \triangleq T \rangle$  to denote the **pattern match** of  $T$  to  $P$

- If the match is successful, it determines bindings of the variables in  $P$  to terms
- Given an expression  $E$ , we write  $E\langle P \triangleq T \rangle$  to denote the term obtained by applying the bindings of the pattern match  $\langle P \triangleq T \rangle$  to the variables in  $E$  with the same names
- If the pattern match fails,  $E\langle P \triangleq T \rangle$  is undefined

Examples:

- $(X + Y)\langle \{X, Y\} \triangleq \{3, 2\} \rangle$  is 5
- $(T ++ [2])\langle [H | T] \triangleq [8] \rangle$  is [2]
- $H\langle [H | T] \triangleq [ ] \rangle$  is undefined

NOTE: The notation  $E\langle P \triangleq T \rangle$  is **not** valid Erlang, but we use it to illustrate Erlang's semantics

# Multiple expressions

Multiple expressions  $E_1, \dots, E_n$  can be combined in a compound expression obtained by separating them using commas

- Evaluating the compound expression entails evaluating all component expressions in the order they appear, and returning the value of the last component expression as the value of the whole compound expression
- A single failing evaluation makes the whole compound expression evaluation fail

Examples:

`3 < 0, 2.`

$\% \text{ evaluates } 3 < 0$   
 $\% \text{ returns } 2$

`3 + true, 2.`

$\% \text{ evaluates } 3 + \text{true}$   
 $\% \text{ fails}$

`R=10, Pi=3.14, 2*Pi*R.`

$\% \text{ binds } 10 \text{ to } R,$   
 $\% \text{ binds } 3.14 \text{ to } Pi$   
 $\% \text{ returns } 62.8\dots$

# Multiple expression blocks

Using **blocks** delimited by **begin...end**, we can introduce **multiple** expressions where **commas** would normally be interpreted in a different way

This may be useful in function calls:

```
power(2, begin X=3, 4*X end)    % returns power(2, 12)
```

Without **begin...end**, the expression would be interpreted as calling a function `power` with three arguments

# List comprehensions

List comprehensions provide a convenient syntax to define lists using pattern matching

It is an expression of the form: [ Expression || P<sub>1</sub> <- L<sub>1</sub>, ..., P<sub>m</sub> <- L<sub>n</sub>, C<sub>1</sub>, ..., C<sub>n</sub> ] where:

- each P<sub>k</sub> is a pattern
  - each L<sub>k</sub> is a list expression
  - each C<sub>k</sub> is a condition (a Boolean expression)
- Intuitively, each pattern P<sub>k</sub> is matched to every element of L<sub>k</sub>, thus determining a binding B
    - if substituting all bound values makes all conditions evaluate to true, the value obtained by substituting all bound values in Expression is accumulated in the list result;
    - otherwise the binding is ignored

Examples:

```
[X*X || X <- [1, 2, 3, 4]]          % is [1, 4, 9, 16]
[X || X <- [1, -3, 10], X > 0]    % is [1, 10]
[{A, B} || A <- [carl, sven], B <- [carlsson, svensson]]
% is [{carl, carlsson}, {carl, svensson},
%       {sven, carlsson}, {sven, svensson}]
```

# Modules

A **module** is a **collection of function definitions** grouped in a file

Modules are the only places where functions can be defined – they cannot directly be defined in the shell

The **main elements** of a module are as follows:

```
-module(foo).    % module with name `foo' in file `foo.erl'  
-export([double/1,up_to_5/0]). % exported functions  
                  % each f/n refers to the function with name `f' and arity `n'  
-import(lists, [seq/2]). % functions imported from module `lists'  
                  % function definitions:  
double(X) -> 2*X.  
up_to_5() -> seq(1, 5).    % uses imported lists:seq
```

**Compiling and using a module in the shell:**

```
1> c(foo).          % compile module `foo' in current directory  
{ok,foo}.           % compilation successful  
2> foo:up_to_5().   % call `up_to_5' in module `foo'  
[1,2,3,4,5]
```

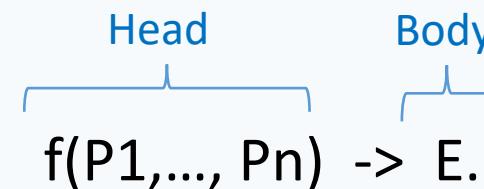
# Function definitions

# Function definitions: basics

In Erlang (and all functional prog. lang.) **functions** are the fundamental units of computation

- A **function** defines how to map values to other values
  - Unlike in imperative programming languages, most functions in Erlang have **no side effects**: they do not change the state of the program executing them (especially their arguments)

The basic definition of an  $n$ -argument function  $f$  (arity  $n$ ), denoted by  $f/n$ , has the form:



- The function **name**  $f$  is an atom
- The function's formal **arguments**  $P_1, \dots, P_n$  are patterns
- The **body**  $E$  is an expression – normally including variables that appear in the arguments

Examples:

<code>identity(X) -&gt; X.</code>	<i>% the identity function</i>
<code>sum(X, Y) -&gt; X + Y.</code>	<i>% the sum function</i>

# Examples of function definitions

The basic definition of an  $n$ -argument function  $f$  (arity  $n$ ), denoted by  $f/n$ , has the form:

$$f(P_1, \dots, P_n) \rightarrow E.$$

More examples:

zero()	$\rightarrow 0.$	$\circledcirc$ integer zero
identity(X)	$\rightarrow X.$	$\circledcirc$ identity
sum(X, Y)	$\rightarrow X + Y.$	$\circledcirc$ sum
head([H _])	$\rightarrow H.$	$\circledcirc$ head
tail([_ T])	$\rightarrow T.$	$\circledcirc$ tail
second({_, Y})	$\rightarrow Y.$	$\circledcirc$ 2nd of pair
positives(L)	$\rightarrow [X \mid X \leftarrow L, X > 0].$	$\circledcirc$ filter positive

# Function call/evaluation

Given the definition of a function  $f/n$ :

$$f(P_1, \dots, P_n) \rightarrow E.$$

a **call expression** to  $f/n$  has the form:

$$f(A_1, \dots, A_n)$$

and is **evaluated** as follows:

1. For each  $1 \leq k \leq n$ , evaluate  $A_k$ , which gives a term  $T_k$
2. For each  $1 \leq k \leq n$ , pattern match  $T_k$  to  $P_k$
3. If all pattern matches are successful, the call expression evaluates to  
 $E(P_1, \dots, P_n \triangleq T_1, \dots, T_n)$
4. Otherwise, the evaluation of the call expression fails

# Examples of function calls

DEFINITIONS	CALLS	VALUE
<code>zero() :-&gt; 0.</code>	<code>zero()</code>	0
<code>identity(X) :-&gt; X.</code>	<code>identity({1,2,3})</code>	{1,2,3}
<code>sum(X, Y) :-&gt; X + Y.</code>	<code>sum(zero(), second({2,3}))</code>	3
<code>head([H _]) :-&gt; H.</code>	<code>head([])</code>	fail
<code>head([H _]) :-&gt; H.</code>	<code>head([3,4,5])</code>	3
<code>tail([_ T]) :-&gt; T.</code>	<code>tail([])</code>	fail
<code>positives(L) :-&gt;</code> <code>[X    X &lt;- L, X &gt; 0].</code>	<code>positives([-2,3,-1,6,0])</code>	[3,6]

# Function definition: clauses

Function definitions can include multiple **clauses**, separated by semicolons:

$$f(P_{11}, \dots, P_{1n}) \rightarrow E_1;$$

$$f(P_{21}, \dots, P_{2n}) \rightarrow E_2;$$

$$\vdots$$

$$f(P_{m1}, \dots, P_{mn}) \rightarrow E_m.$$

A **call expression** is evaluated against each clause in textual order; the first successful *match* is returned as the result of the call

Therefore, we should enumerate clauses from more to less specific

```
lazy_or(true, true)  -> true;
lazy_or(_, true)    -> true;
lazy_or(_, _)       -> false.
```

This function does not work as expected  
unless this clause is listed last

# Pattern matching with records

Pattern matching an expression  $R$  of record type  $\text{rec}$

$$\#_{\text{rec}}\{f_1=P_1, \dots, f_n=P_n\} = R$$

succeeds if, for all  $1 \leq k \leq n$ , field  $f_k$  in  $R$ 's evaluation (i.e.,  $R\#\text{name}.f_k$ ) matches to pattern  $P_k$

If record type  $\text{rec}$  has fields other than  $f_1, \dots, f_n$ , they are ignored in the match

Thanks to this behavior, using arguments of record type provides a simple way to extend data definitions without having to change the signature of all functions that use that datatype

# Flexible arguments with records: Example

```
-record(error, {code}).  
error_message(#error{code=100}) -> io.format("Wrong address");  
error_message(#error{code=101}) -> io.format("Invalid username");  
...  
error_message(_) -> io.format("Unknown error").
```

If we want to add more information to the type `error`, we only have to change the record definition, and the clauses using the new information:

```
-record(error, {code, line_number}).  
error_message(#error{code=100}) -> io.format("Wrong address");  
error_message(#error{code=101}) -> io.format("Invalid username");  
...  
error_message(#error{code=C, line_number=L}) -> io.format("Unknown error p", [C, L]).
```

Compare this to the case where we would have had to change `error_message` from a unary to a binary function!

# Function definition: guards

Clauses in function definitions can include any number of **guards** (also called **conditions**):

```
f(Pk1, . . . , Pkn) when Ck1, Ck2, . . . -> Ek;
```

A guarded clause is selected only if **all guards**  $C_{k1}, C_{k2}, \dots$  evaluate to **true** under the match, that is if  $C_{ki}(P_{k1}, \dots, P_{kn} \triangleq T_{k1}, \dots, T_{kn})$  evaluates to true for all guards  $C_{ki}$  in the clause

More generally, two guards can be separated by either a comma or a semicolon: **commas** behave like lazy **and** (both guards have to hold); **semicolon** behave like lazy **or** (at least one guard has to hold)

```
can_drive(Name, Age) when Age >= 18 -> Name ++ " can drive";
can_drive(Name, _) -> Name ++ " cannot drive".
```

```
same_sign(X, Y) when X > 0, Y > 0; X < 0, Y < 0 -> true;
same_sign(_, _) -> false.
```

## Type checking -- at runtime

Since Erlang is dynamically typed, there are cases where we have to **test** the **actual type** of an expression

- For example, because a certain operation is only applicable to values of a certain type

To this end, Erlang provides several **test functions** whose names are self-explanatory:

```
is_atom/1
is_boolean/1
is_float/1
is_integer/1
is_list/1
is_number/1
is_pid/1
is_port/1
is_tuple/1
```

Use these only when necessary: in most cases defining implicitly partial functions is enough

## Function definition: local binding

The expression **body** in a function definition can include **compound expressions with bindings**:

$$f(Pk_1, \dots, Pk_n) \rightarrow V_1=E_1, \dots, V_w=E_w, E_k;$$

Such bindings are **only visible** within the function definition

They are useful to define shorthands in the definition of complex expressions

```
volume({cylinder, Radius, Height}) ->
  Pi=3.1415,
  BaseArea=Pi*Radius*Radius,
  Volume=BaseArea*Height,
  Volume.
```

# If expressions (guard patterns)

**Ifs** provide a way to express conditions alternative to guards (in fact, **ifs** are called – somewhat confusingly – *guard patterns* in Erlang)

An **if expression**:

```
if
    C1 -> E1;
    :
    Cm -> Em
end
```

evaluates to the expression  $E_k$  of the first guard  $C_k$  in **textual order** that evaluates to true; if no guard evaluates to true, evaluating the **if** expression fails

```
age(Age) ->
    if Age > 21 -> adult;
       Age > 11 -> adolescent;
       Age > 2 -> child;
       true        -> baby end.
```

# Case expressions

Cases provide an additional way to use pattern matching to define expressions. A case expression:

```
case E of
  P1 -> E1;
  :
  Pm -> Em
end
```

evaluates to  $E_k \langle P_k \triangleq T \rangle$ , where  $E$  evaluates to  $T$ , and  $P_k$  is the first pattern in textual order that  $T$  matches to; if  $T$  matches no pattern, evaluating the case expression fails

Patterns may include when clauses, with the same meaning as in function definitions

```
years(X) ->
  case X of {human, Age} -> Age;
             {dog, Age}    -> 7*Age;
             _                  -> cant_say
end.
```

# Which one should I use?

Having several [different ways](#) of defining a function can be confusing. There are no absolute rules, but here are some [guidelines](#) that help you write idiomatic code:

- the **first** option to try is using [pattern matching](#) directly in a function's arguments, using different clauses for different cases
- if parts of a pattern expression depend on others, you may consider using [case expressions](#) to have nested patterns
- you do not need [if expressions](#) very often (but it's good to know what they mean, and sometimes they may be appropriate)

# Recursion

# Recursion in programming

- Recursion is a style of programming where functions are defined in terms of themselves

The **definition** of a function  $f$  is **recursive** if it includes a call to  $f$  (directly or indirectly)

```
% compute X^n
power(X, 0) -> 1;
power(X, N) -> X * power(X, N-1).
```



Recursive call

# Recursion in mathematics

Recursion is a **style of programming** where functions are defined in terms of themselves

The **definition** of a function  $f$  is **recursive** if it includes a call to  $f$  (directly or indirectly)

Definition of **natural numbers**:

- $0$  is a natural number;
- if  $n$  is a natural number then  $n + 1$  is a natural number.



Recursive/inductive definition

# Recursion: from math to programming

Recursion in programming provides a natural way of implementing recursive definitions in mathematics

Factorial of a nonnegative integer  $n$ :

$$n! = n \cdot (n - 1) \dots 1 = n \cdot (n - 1) \dots 1$$

$\underbrace{\hspace{10em}}_{n \text{ terms}}$        $\underbrace{\hspace{10em}}_{n-1 \text{ terms}}$

$$n! = \begin{cases} 1 & \text{if } 0 \leq n \leq 1 \\ n \cdot (n-1)! & \text{if } n > 1 \end{cases}$$

Base case  
 Recursive/inductive case

# Recursion: from math to programming

Recursion in programming provides a natural way of implementing recursive definitions in mathematics

Factorial of a nonnegative integer  $n$ :

$$n! = \begin{cases} 1 & \text{if } 0 \leq n \leq 1 \\ n \cdot (n-1)! & \text{if } n > 1 \end{cases}$$

Base case      Recursive/inductive case

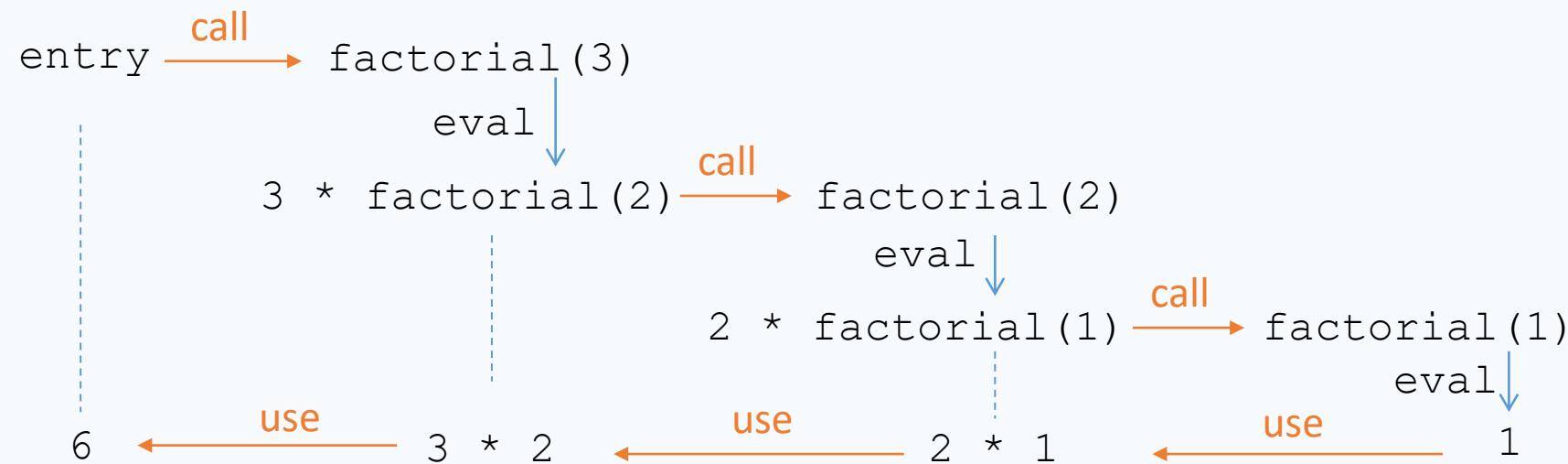
```
factorial(N) when N <= 1 -> 1;                                   % base case
factorial(N)                                                        -> N *factorial(N-1).                                   % recursive case
```

↑  
Recursive call

# How does recursion work?

Each recursive call triggers an **independent evaluation** of the recursive function  
(Independent means that it works on its own private copy of actual argument expressions)

When a recursive instance terminates evaluation, its value is used in the calling instance **for its own evaluation**



# Recursion as a design technique

Recursion as a programming technique is useful to design programs using the **divide and conquer** approach:

To solve a problem instance  $P$ , split  $P$  into problem instances  $P_1, \dots, P_n$  chosen such that:

1. Solving  $P_1, \dots, P_n$  is simpler than solving  $P$  directly
2. The solution to  $P$  is a simple combination of the solutions to  $P_1, \dots, P_n$

In functional programming, recursion goes hand in hand with pattern matching:

- Pattern matching splits a function argument's into smaller bits according to the input's structure
- Recursive function definitions define the base cases directly, and combine simpler cases into more complex ones

# Recursive functions: Sum of list

Define a function `sum(L)` that returns the sum of all numbers in `L`

1. The base case (the simplest possible) is when `L` is empty:  $\text{sum}([]) \rightarrow 0$
2. Let now `L` be non-empty: a non empty list matches the pattern `[H|T]`
  - `H` is a single number, which we must add to the result
  - `T` is a list, which we can sum by calling `sum` recursively

```
sum([])      -> 0;           % base case
sum([H|T])  -> H + sum(T). % recursive case
```

Can we switch the  
order of clauses?  
In this case, YES

To make the function more robust, we can skip over all non-numeric elements:

```
sum([])          -> 0;           % base case
sum([H|T]) when is_number(H) -> H + sum(T); % recursive case 1
sum([_|T])       -> sum(T).    % recursive case 2
```

# Recursive functions: Last list element

Define a function `last(L)` that returns the **last element** of `L`

1. When `L` is empty, `last` is undefined, so we can ignore this case
2. The simplest case is then when `L` is one element:  $\text{last}([\text{E}]) \rightarrow \text{E}$
3. Let now `L` be non-empty: a non empty list matches the pattern `[H | T]`
  - `E` is the first element, which we throw away
  - `T` is a list, whose last element we get by calling `last` recursively

$\text{last}([\text{E}]) \rightarrow \text{E};$       %; base case  
 $\text{last}([\underline{\_} | \text{T}]) \rightarrow \text{last}(\text{T}).$       %; recursive case

Can `T` match the empty list?

No, because neither of the clauses match the empty list

To make this explicit, we could write:

$\text{last}([\text{E} | []]) \rightarrow \text{E};$       %; base case  
 $\text{last}([\underline{\_} | \text{T}]) \rightarrow \text{last}(\text{T}).$       %; recursive case

# Tail recursion

A recursive function  $f$  is **tail recursive** if the evaluation of  $f$ 's body evaluates the recursive call **last**

```
% general recursive:  
power(_, 0) ->  
    1;  
power(X, N) ->  
    X * power(X, N-1).
```

```
% tail recursive:  
power(X, N) ->  
    power(X, N, 1).  
power(_, 0, Accumulator) ->  
    Accumulator;  
power(X, N, Accumulator) ->  
    power(X, N-1, X*Accumulator).
```

Overloading:

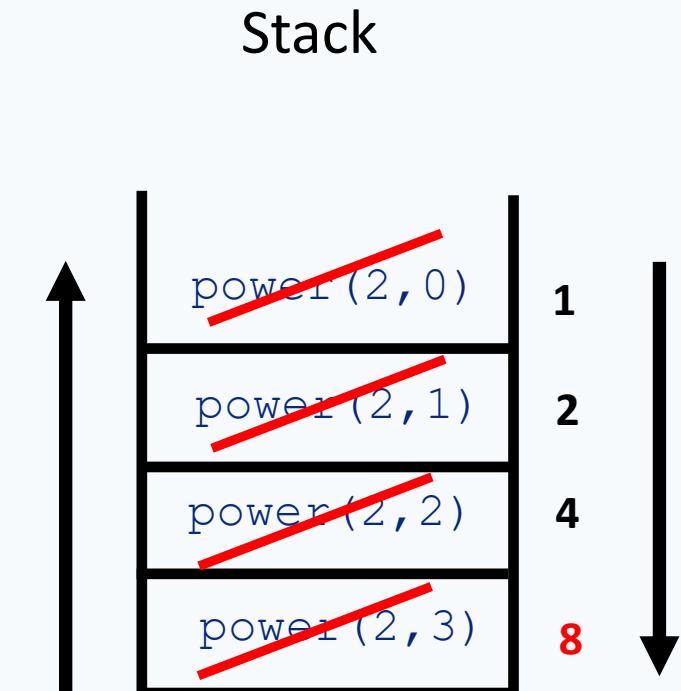
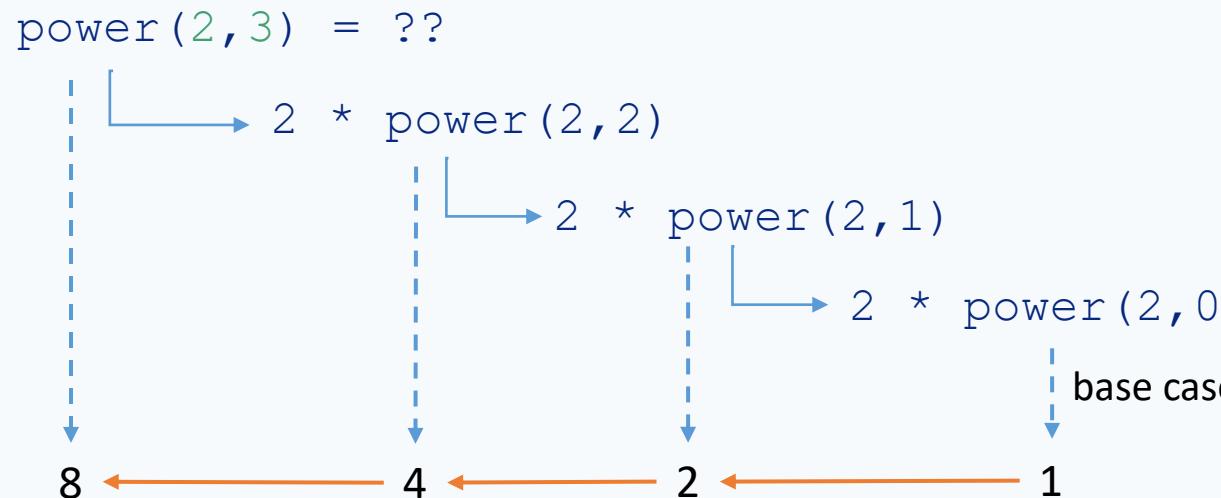
two functions **power/2** and **power/3**

- Tail-recursive functions are generally more efficient than general-recursive functions
- When efficiency is **not an issue**, there is no need to use a tail-recursive style; but we will use tail-recursive functions extensively (and naturally) when implementing servers

# General Recursion vs Tail Recursion

## General recursion:

```
% general recursive:
power(_, 0) -> 1;
power(X, N) -> X * power(X, N-1).
```



# General Recursion vs Tail Recursion

## Tail recursion:

% tail recursive:

```
power(X, N) -> power(X, N, 1).
power(_, 0, Accumulator) -> Accumulator;
power(X, N, Accumulator) -> power(X, N-1, X*Accumulator).
```

power(2, 3) = ??

  └→ power(2, 3, 1)

    └→ power(2, 2, 2\*1)

      └→ power(2, 1, 2\*2)

      └→ power(2, 0, 2\*4)

      └ base case

8

Stack



# Impure and higher-order functions

# Where are all the statements, assignments, loops?

Statements, assignments, and loops are not available as such in Erlang

Everything is an expression that gets evaluated:

- (Side-effect free) expressions are used instead of statements
- (Pure) functions return modified copies of their arguments instead of modifying the arguments themselves
- One-time bindings are used instead of assignments that change values to variables
- Recursion is used instead of loops

The sparse presence of side effects helps make functional programs higher level than imperative ones

# Printing to screen

The expressions we have used so far have no **side effects**, that is they do not change the state but simply evaluate to a value

- Not all expressions are side-effect free in Erlang
  - **Input/output** is an obvious exception: to print something to screen, we **evaluate** an expression call, whose side effect is printing

`io:format(Format, Data)` % *print the string Format, interpreting control sequences on Data*

CONTROL SEQUENCE	DATA	
<code>~B</code>	integer	
<code>~g</code>	float	
<code>~s</code>	string	
<code>~p</code>	any Erlang term	
<code>~n</code>	line break	
		You can use <code>fwrite</code> instead of <code>format</code>

```
1> io:format("~s ~B. ~p~n~s ~B~n", ["line", 1, true, "line", 2]).  
line 1. true  
line 2
```

# Exception handling

Erlang has an **exception handling mechanism** that is similar to a functional version of Java's try/catch/finally blocks:

```
try Expr of
    Success1 -> Expr1;
    ...
catch
    Error1:Fail1 -> Recov1;
    ...
after After end
```

- The `try` blocks behaves like a `case` block
- If evaluating `Expr` raises an exception, it gets pattern matched against the clauses in `catch` (`Errork`'s are error types, `Failk`'s are patterns, and `Recovk`'s are expressions)
- Expression `After` in the `after` clause always gets evaluated in the end (but does not return any value: used to close resources)

# Exception handling: Example

Function `safe_plus` tries to evaluate the sum of its arguments:

- if evaluation succeeds, it returns the result
- if evaluation raises a `badarith` exception, it returns `false`

```
safe_plus(X, Y) ->
  try X + Y of
    N -> N
  catch
    error:badarith -> false
  end.
```

Example of using it:

```
1> safe_plus(2, 3).
5
```

```
2> safe_plus(2, []).
false
```

# Functions are values too

Functions are first-class objects in Erlang: they can be passed around like any other values, and they can be arguments of functions

- A function  $f/k$  defined in module  $m$  is passed as argument `fun m:f/k`

This makes it easy to define functions that apply other functions to values following a pattern

```
% apply function F to all elements in list L
map(F, [])    -> [];
map(F, [H|T]) -> [F(H) | map(F, T)].
```

```
1> map(fun m:age/1, [12, 1, 30, 56]).
```

[adolescent, baby, adult, adult]

```
age(Age) ->
  if Age > 21 -> adult;
     Age > 11 -> adolescent;
     Age > 2 -> child;
     true        -> baby end.
```

A function that takes another function as argument is called **higher-order**

# High-Order Functions

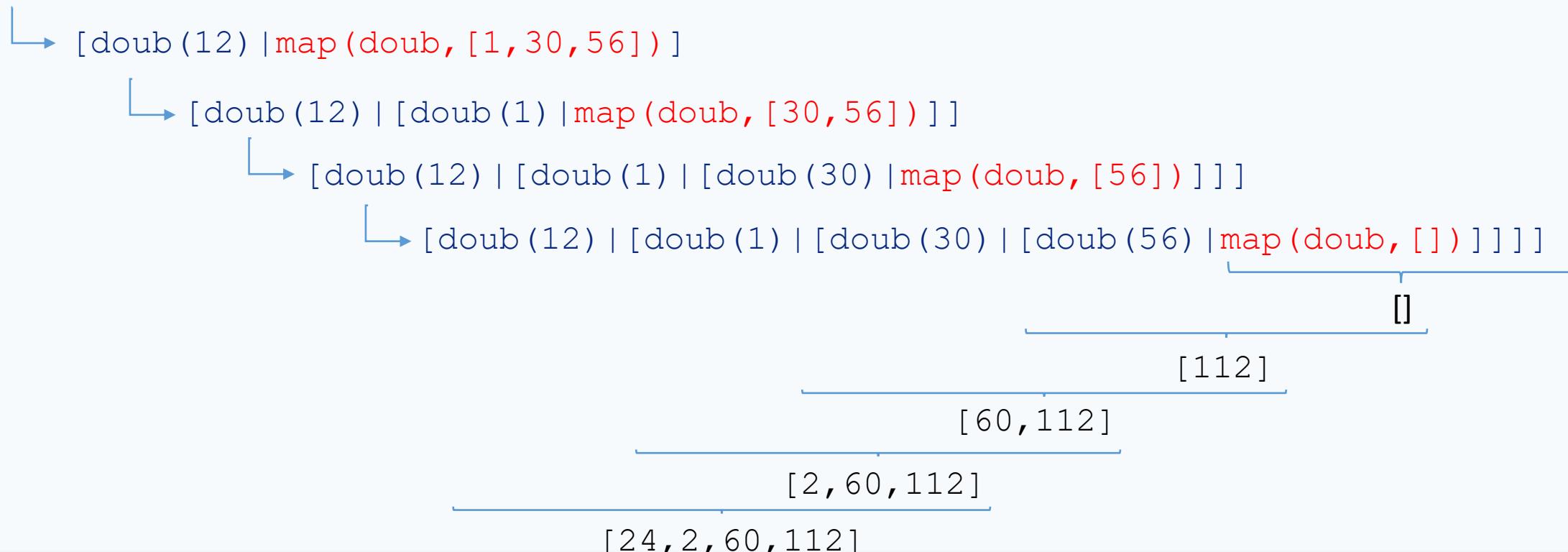
% apply function F to all elements in list L  
`map(F, []) -> [];`  
`map(F, [H|T]) -> [F(H) | map(F, T)].`

Let's define a function:

`doub(X) -> 2*X;`

What is the result of calling `map (doub/1, [12,1,30,56])` ?

`map (doub/1, [12,1,30,56]) = ??`



# Inline functions

Sometimes it is necessary to **define** a function **directly in an expression** where it is used

For this we can use **anonymous functions** – also called lambdas, closures, or funs (the last is Erlang jargon):

```
fun
    (A1) -> E1;
    :
    (An) -> En
end
```

where each  $A_k$  is a sequence of patterns, and each  $E_k$  is a body

```
% double every number in the list
1> map(fun (X)->2*X end, [12, 1, 30, 56]).  
[24,2,50,112]
```

# Working on lists

Module `lists` includes many useful predefined functions to work on lists

These are some you should know about – but check out the full module documentation at  
<http://erlang.org/doc/man/lists.html>:

<code>all(Pred, List)</code>	<i>% do all elements E of List satisfy Pred(E) ?</i>
<code>any(Pred, List)</code>	<i>% does any element E of List satisfy Pred(E) ?</i>
<code>filter(Pred, List)</code>	<i>% all elements E of List that satisfy Pred(E)</i>
<code>last(List)</code>	<i>% last element of List</i>
<code>map(Fun, List)</code>	<i>% apply Fun to all elements of List</i>
<code>member(Elem, List)</code>	<i>% is Elem an element of List?</i>
<code>reverse(List)</code>	<i>% List in reverse order</i>
<code>seq(From, To)</code>	<i>% list [From, From+1, ..., To]</i>
<code>seq(From, To, I)</code>	<i>% list [From, From+I, ... , To]</i>

# Folds

Several functions compute their result by recursively accumulating values from a list:

```
sum([])    -> 0;
sum([H|T]) -> H + sum(T).
```

```
len([])    -> 0;
len([H|T]) -> 1 + len(T).
```

We can generalize this pattern into a single higher-order function `fold(F, R, L)`: starting from an **initial value** `R`, combine all elements of **list** `L` using **function** `F` and accumulate the result:

```
fold(_, Result, [])    -> Result;
fold(F, Result, [H|T]) -> F(H, fold(F, Result, T)).
```

Using `fold`, we can define `sum` and `len`:

```
sum(L) ->
  fold(fun (X, Y) -> X+Y end, 0, L).
```

```
len(L) ->
  fold(fun (X, Y) -> 1+Y end, 0, L).
```

Erlang module `lists` offers functions `foldr/3` (which behaves like our `fold`) and `foldl/3` (a tail-recursive version of `fold`, with the same arguments)

# Folds: Example

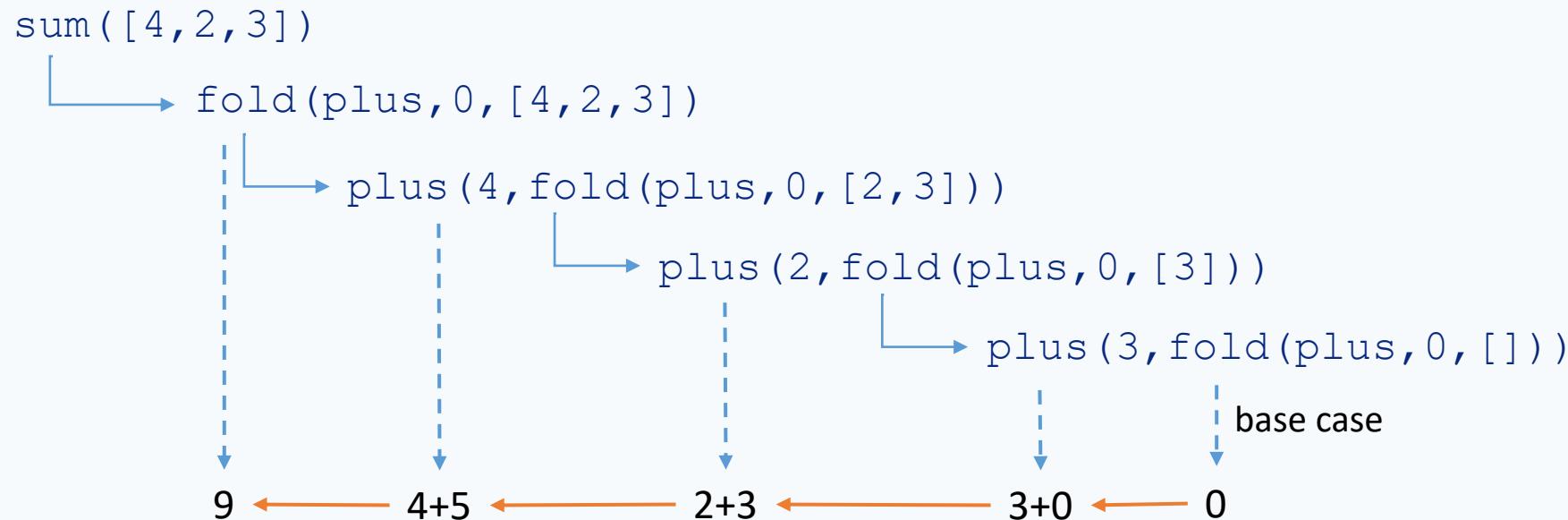
```
fold(_, Result, [])      -> Result;
fold(F, Result, [H|T])  -> F(H, fold(F, Result, T)).
```

Let's call this function plus

Let's define a sum of a list using fold

```
sum(L) -> fold( fun(X,Y)-> X+Y end, 0, L )
```

Let's try it!



# Message-Passing Concurrency in Erlang

Lecture 7 of TDA384/DIT391

Principles of Concurrent Programming



UNIVERSITY OF  
GOTHENBURG

Nir Piterman and Gerardo Schneider

Chalmers University of Technology | University of Gothenburg



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY

# Lesson's menu

- Actors and message passing
- Sending and receiving messages
- Stateful processes
- Clients and servers
- Generic servers
- Location transparency & distribution

# What is Erlang?

Erlang combines a functional language with message-passing features:

- The functional part is sequential, and is used to define the behavior of processes
- The message-passing part is highly concurrent: it implements the actor model, where actors are Erlang processes

This lecture covers the message-passing/concurrent part of Erlang

# ACTORS AND MESSAGE PASSING

# Erlang's Principles

Concurrency is fundamental in Erlang, and it follows models that are quite different from those offered by most imperative languages

In Erlang (from Armstrong's PhD thesis):

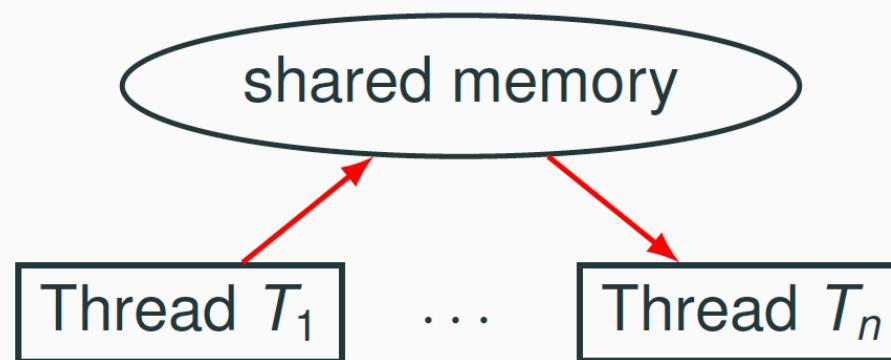
- Processes are strongly isolated
- Process creation and destruction is a lightweight operation
- Message passing is the only way for processes to interact
- Processes have unique names
- If you know the name of a process, you can send it a message
- Processes share no resources
- Error handling is non-local
- Processes do what they are supposed to do or fail

Compare these principles to programming using Java threads!

# Shared Memory vs. Message Passing

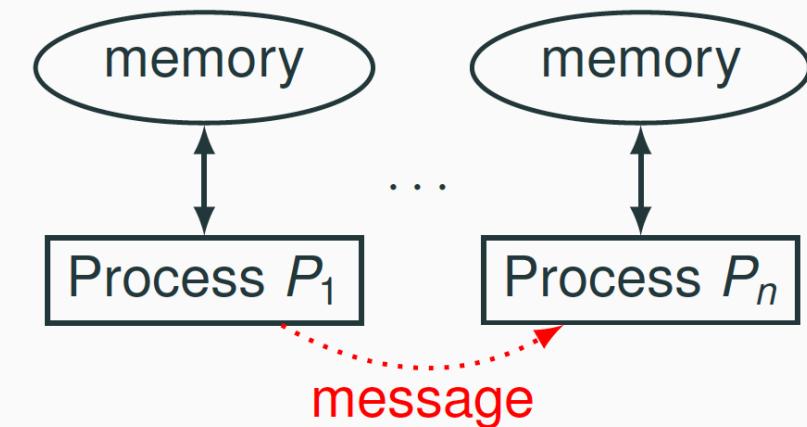
## Shared memory:

- synchronize by writing to and reading from shared memory
- natural choice in shared memory systems such as threads



## Message passing:

- synchronize by exchanging messages
- natural choice in distributed memory systems such as processes



# The Actor Model

Erlang's message-passing concurrency mechanisms implement the **actor model**:

- **Actors** are abstractions of processes
- **No shared state** between actors
- Actors **communicate** by **exchanging** messages – asynchronous message passing

A metaphorical **actor** is an “active agent which plays a role on cue according to a script” (Garner & Lukose, 1989)

# Actor and Messages

Each actor is identified by an **address**

An **actor** can:

- **send** (finitely many) **messages** to other actors via their addresses
- **change** its **behavior** – what it computes, how it reacts to messages
- **create** (finitely many) **new actors**

A **message** includes:

- a **recipient** – identified by its address
- **content** – arbitrary information

# The Actor Model in Erlang

The entities in the actor model correspond to **features of Erlang** (possibly with some terminological change)

ACTOR MODEL	Erlang	LANGUAGE
actor	sequential process	
address	PID (process identifier)	pid type
message	an Erlang term	{From, Content}
behavior	(defined by) functions	
create actor	spawning	spawn
dispose actor	termination	
send message	send expression	To ! Message
receive message	receive expression	receive...end

# SENDING AND RECEIVING MESSAGES

# A Process's Life

A **process**:

- is **created** by calling `spawn`
- is identified by a **pid** (process identifier)
- **executes** a function (passed as argument to `spawn`)
- when the function terminates, the process **ends**

# The spawn function

Function `spawn(M, F, Args)` creates a `new process`:

- the process runs function F in module M with arguments Args
- evaluating `spawn` returns the pid of the created process

Within a process's code, function `self()` returns the process's `pid`

Within a module's code, macro `?MODULE` gives the `module's name`

Calling `spawn (fun () -> f (a1, ..., an) end)` is equivalent to

`spawn (?MODULE, f, [a1, ..., an])` but does not require exporting f

# Processes: Examples

A process code:

```
-module(procs).

print_sum(X,Y) ->
  io:format("~p~n", [X+Y]). 

compute_sum(X,Y) -> X + Y.
```

```
3> spawn(fun ()-> true end).
<0.82.0> % pid of spawned process
4> self().
<0.47.0> % pid of process running shell
```

Creating processes in the shell:

```
3> spawn(procs, print_sum, [3, 4]).
7          % printed sum
<0.78.0> % pid of spawned process

2> spawn(procs, compute_sum, [1, 7]).
<0.80.0> % pid of spawned process
          % result not visible!
```

# Sending Messages

A **message** is any **term** in Erlang

Typically, a message is the result of **evaluating** an expression

The expression

Pid ! Message

"Bang" operator



sends the evaluation  $T$  of **Message** to the process with pid **Pid**; and returns  $T$  as result

**Bang** is right-associative

To send a message to multiple recipients, we can combine multiple bangs:

Pidn1 ! Pidn2 ! ... ! Pidn ! Message

# Mailboxes

Every process is equipped with a **mailbox**, which behaves like a FIFO **queue** and is filled with the **messages** sent to the process in the order they arrive.

Mailboxes make **message-passing asynchronous**: the sender does not wait for the recipient to receive the message; messages queue in the mailbox until they are processed

To check the content of process Pid's mailbox, use functions:

- `process_info(Pid, message_queue_len)`: how many elements are in the mailbox
- `process_info(Pid, messages)`: list of messages in the mailbox (oldest to newest)
- `flush()`: empty the current process's mailbox

```
1> self() ! self() ! hello.    % send 'hello' twice to self
2> self() ! world.           % send 'world' to self
3> erlang:process_info(self(), messages)
{messages, [hello, hello, world]}    % queue in mailbox
```

# Receiving messages

To **receive messages**, use the **receive** expression:

```
receive
    P1 when C1 -> E1;
    :
    Pn when Cn -> En
end
```

Evaluating the **receive** expression selects the **oldest** term  $T$  in the receiving process's mailbox that matches a pattern  $P_k$  and satisfies condition  $C_k$

If a term  $T$  that matches exists, the **receive** expression evaluates to  $E_k \langle P_k \triangleq T \rangle$ ; otherwise, evaluation **blocks** until a suitable message arrives

# The receiving algorithm

How evaluating **receive** works, in pseudo-code:

```
Term receive(Queue<Term> mailbox, List<Clause> receive) {  
    while (true) {  
        await(!mailbox.isEmpty()); // block if no messages  
        for (Term message: mailbox) // oldest to newest  
            for (Clause clause: receive) // in textual order  
                if (message.matches(clause.pattern))  
                    // apply bindings of pattern match  
                    // to evaluate clause expression  
                return clause.expression <clauses.pattern△message>;  
    }  
}
```

# Receiving messages: examples

A simple echo function, which prints any message it receives:

```
echo() ->
  receive Msg -> io:format("Received: ~p~n", [Msg]) end.
```

Sending messages to echo in the shell:

```
1> Echo=spawn(echo, echo, []).
% now Echo is bound to echo's pid
2> Echo ! hello. % send 'hello' to Echo
Received: hello % printed by Echo
```

To make the receiving process permanent, it calls itself after receiving:

```
repeat_echo() ->
  receive Msg -> io:format("Received: ~p~n", [Msg]) end,
  repeat_echo(). % after receiving, go back to listening
```

tail recursive, thus no memory consumption problem!

# Message delivery order

Erlang's runtime only provides weak guarantees of **message delivery order**:

- If a process S sends some messages to **another process** R, then R will receive the messages in the **same order** S sent them
- If a process S sends some messages to **two (or more)** other processes R and Q, there is **no guarantee** about the order in which the messages sent by S are received by R relative to when they are received by Q

In practice, pretty much all the Erlang code we will write does **not rely on any assumptions** about message delivery order

Even defining – let alone enforcing – an absolute time across multiple independent processes (which could even be geographically distributed) would be tricky: in order to synchronize, processes can only exchange messages!

# Message delivery order: single process

If **process S** sends messages a,b,c – in this order – to **process R**, then **R** will receive them in its mailbox in the **same order**

sender process S:

```
R ! a,  
R ! b,  
R ! c.
```

receiver process R:

R's mailbox: 

R is process R's pid

# Message delivery order: multiple processes

If process S sends messages a,b,c – in this order – to process R and to process Q, then R and Q may receive them in any order relative to each other.

Possible scenarios:

sender process S:

R ! a,  
Q ! b,  
Q ! c.



Q is process Q's pid

receiver process R:

R's mailbox: 

a
---

receiver process Q:

Q's mailbox: 

b	c
---	---

# Stateful processes

# A ping server

A **ping server** is constantly listening for requests; to every message ping, it replies with a message ack sent back to the sender.

In order to **identify the sender**, it is customary to encode messages as tuples of the form:

{ SenderPid, Message }

```
ping() -> receive
    {From, ping} -> From ! {self(), ack};   % send ack to pinger
                           -> ignore          % ignore any other message
    end, ping().           % next message
```

## Combining the echo and ping servers:

```
1> Ping = spawn(echo, ping, []), Echo = spawn(echo, repeat_echo, []).
2> Ping ! {Echo, ping}.    % send ping on Echo's behalf
Received: {<0.64.0>, ack} % ack printed by Echo
3> Ping ! {Echo, other}.  % send other message to Ping
% no response
```

# Stateful processes

Processes can only operate on the arguments of the function they run, and on whatever is sent to them via message passing

- Thus, we store **state** information using **arguments**, whose value gets updated by the **recursive calls** used to make a process permanently running

A **stateful process** can implement the message-passing analogue of the **concurrent counter** that used Java threads

The Erlang counter function recognizes two commands, sent as messages:

- increment: add one to the stored value
- count: send back the currently stored value

```
base-counter(N) ->
    receive {From, Command} -> case Command of
        increment -> base-counter(N+1);           % increment counter
        count      -> From ! {self(), N},          % send current value
                         base-counter(N);       % do not change value
        U          -> io:format("?", [U])        % unrecognized command
    end end.
```

# Concurrent counter: first attempt

```
base-counter(N) ->
  receive {From, Command} -> case Command of
    increment -> base-counter(N+1);                                % increment counter
    count      -> From ! {self(), N},                                % send current value
                     base-counter(N);                                % do not change value
    U          -> io:format("?", ~p~n, [U])                         % unrecognized command
  end end.
```

Evaluated only when spawning a process running FCount

```
increment-twice() ->
  Counter = spawn(counter, base-counter, [0]),                      % counter initially 0
  % function sending message 'increment' to Counter:
  FCount = fun () -> Counter ! {self(), increment} end,
  spawn(FCount), spawn(FCount),                                         % two procs running FCount
  Counter ! {self(), count},                                            % send message 'count'
  % wait for response from Counter and print it
  receive {Counter, N} -> io:format("Counter is: ~p~n", [N])
end.
```

# Concurrent counter: first attempt (cont'd)

Running `increment_twice()` does not seem to behave as expected:

```
1> increment_twice().
```

Counter **is**: 0

The problem is that there is **no guarantee** that the **message delivery order** is the same as the sending order: the request for count may be delivered before the two requests for increment (or even before the two processes have sent their increment requests).

A temporary workaround is **waiting some time** before asking for the count, hoping that the two increment messages have been delivered:

`wait_and_hope() ->`

```
Counter = spawn(counter, base_counter, [0]),                      % counter initially 0
FCount = fun () -> Counter ! {self(), increment} end,
spawn(FCount), spawn(FCount),                                     % two processes running FCount
timer:sleep(100),                                              % wait for 'increment' 2b delivered
Counter ! {self(), count},                                         % send message 'count'
receive {Counter, N} -> io:format("Counter is: ~p~n", [N])
end.
```

# Synchronization in an asynchronous world

Since there is **no guarantee** that the **message delivery order** is the same as the sending order when multiple processes are involved, the only robust mechanism for synchronization is **exchanging messages** following a suitable **protocol**

For example, the counter sends **notifications** of every update to a monitoring process:

```
counter(N, Log) ->  
  
receive  
    {_, increment} ->  
        Log ! {self(), N+1}, % send notification  
        counter(N+1, Log); % update count  
    {From, count} -> % send count, next message  
        From ! {self(), N}, counter(N, Log)  
end.
```

# Concurrent counter with monitoring process

```
counter(N, Log) ->
```

**receive**

```
{-, increment} -> Log ! {self(), N+1}, % send notification
                           counter(N+1, Log); % update count
{From, count} -> From ! {self(), N}, counter(N, Log) % send count, next message
end.
```

% set up counter and incrementers; then start monitor:

```
Increment-and-monitor() ->
```

```
Counter = spawn(?MODULE, counter, [0, self()]),
```

```
FCount = fun () -> Counter ! {self(), increment} end,
```

```
spawn(FCount), spawn(FCount),
```

```
monitor_counter(Counter). % start monitor
```

% send notification

% update count

% send count, next message

Spawns a process from this module and executes function counter with parameters N=0 and Log=self()  
(PiD of the spawned process)

FCount sends a message to the recently created process (Counter) with self as parameter and a call to increment

```
monitor-counter(Counter) ->
```

**receive**

```
{Counter, N} -> io:format("Counter is: ~p~n", [N])
```

**end,**

```
monitor_counter(Counter).
```

What happens to messages not in this format?

They stay in the mailbox!

In the shell: counter:increment\_and\_monitor().

You will get:

Counter is: 1

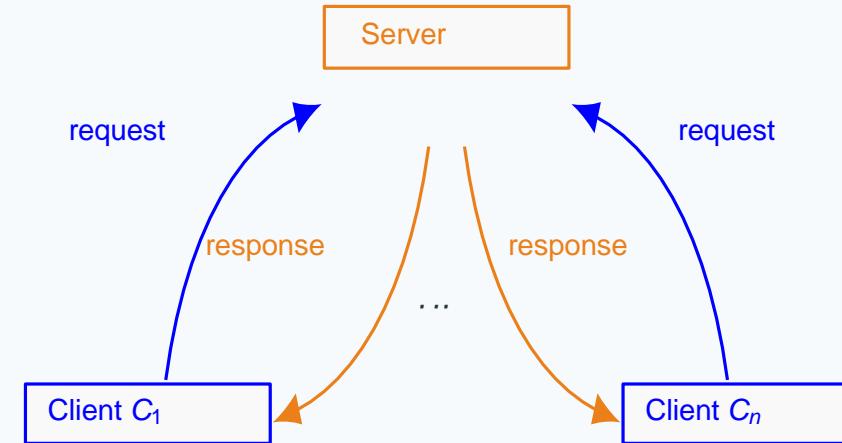
Counter is: 2

# Clients and servers

# Client/server communication

The **client/server architecture** is a widely used communication model between processes using message passing:

1. A **server** is available to serve requests from any clients
2. An arbitrary number of **clients** send commands to the server and wait for the server's response



Many **Internet services** (the web, email, . . .) use the client/server architecture

# Servers

A **server** is a process that:

- responds to a fixed number of **commands** – its **interface**
- runs **indefinitely**, serving an arbitrary number of **requests**, until it receives a shutdown command
- can serve an **arbitrary** number of **clients** – which issue commands as **messages**

Each command is a **message** of the form:

$$\{\text{Command}, \text{From}, \text{Ref}, \text{Arg1}, \dots, \text{Argn}\}$$

- **Command** is the command's name
- **From** is the pid of the client issuing the command
- **Ref** is a unique identifier of the request (so that clients can match responses to requests)
- **Arg1, ..., Argn** are arguments to the command

Each command is **encapsulated in a function**, so that clients need not know the structure of messages to issue commands

# A math server

The **interface** of a math server consists of the following **commands**:

**factorial(M)**: compute the factorial of  $M$

**status()**: return the number of requests served so far (without incrementing it)

**stop()**: shutdown the server

We build an Erlang **module** with interface:

**start()**: start a math server, and return the server's pid

**factorial(S,M)**: compute factorial of  $M$  on server with pid  $S$

**status(S)**: return number of requests served by server with pid  $S$

**stop(S)**: shutdown server with pid  $S$

```
-module(math_server).  
-export([start/0,factorial/2,status/1,stop/1]).
```

# Math server: event loop

```
loop(N) ->
    receive
        % 'factorial' command:
        {factorial, From, Ref, M} ->
            From ! {response, Ref, compute_factorial(M)},
            loop(N+1);                                % increment request number
        % 'status' command:
        {status, From, Ref} ->
            From ! {response, Ref, N},
            loop(N);                                % don't increment request number
        % 'stop' command:
        {stop, _From, _Ref} ->  ok
    end.
```

Ordinary Erlang function computing factorial

```
compute_factorial(M)
```

% increment request number

% don't increment request number

This function needs **not** be exported, unless it is spawned by another function of the module using `spawn(?MODULE, loop, [0])`  
(In that case, it's called via its module, so it must be exported)

# Math server: starting and stopping

We start the server by spawning a process running `loop(0)`:

```
% start a server, return server's pid
start() ->
    spawn(fun () -> loop(0) end).
```

We shutdown the server by sending a command `stop`:

```
% shutdown 'Server'
stop(Server) ->
    Server ! {stop, self(), 0}, % Ref is not needed
ok.
```

# Math server: factorial and status

We compute a factorial by sending a command factorial:

```
% compute factorial(M) on 'Server':
factorial(Server, M) ->
  Ref = make-ref(), % unique reference number
  % send request to server:
  Server ! {factorial, self(), Ref, M},
  % wait for response, and return it: pid of process calling factorial
  receive {response, Ref, Result} -> Result end.
```

Returns a number that is unique among connected nodes  
in the system

pid of process calling factorial

We get the server's status by sending a command status:

```
% return number of requests served so far by 'Server':
status(Server) ->
  Ref = make-ref(), % unique reference number
  % send request to server:
  Server ! {status, self(), Ref},
  % wait for response, and return it:
  receive {response, Ref, Result} -> Result end.
```

# Math server: clients

After creating a server instance, clients simply interact with the server by calling functions of module `math-server`:

```
1> Server = math-server:start().  
<0.27.0>  
2> math-server:factorial(Server, 12).  
479001600  
3> math-server:factorial(Server, 4).  
24  
4> math-server:status(Server).  
2  
5> math-server:status(Server).  
2  
5> math-server:stop(Server). ok  
6> math-server:status(Server).  
% blocks waiting for response
```

# Generic servers

# Generic servers

A **generic server** takes care of the communication patterns behind every server

Users instantiate a generic server by providing a suitable **handling function**, which implements a specific server functionality

A generic server's **start** and **stop** functions are almost identical to the math server's – the only difference is that the event loop also includes a handling function:

```
start(InitialState, Handler) ->  
spawn(fun () -> loop(InitialState, Handler) end).
```

Used to receive the “concrete” server implementation we want this generic server to instantiate

```
stop(Server) ->  
Server ! {stop, self(), 0}, % Ref is not needed  
ok.
```

Handler is a function that implements, e.g., all the different operations a Math server might do

# Generic servers: event loop

The generic server's **event loop** has its current state and the handling function as arguments:

```
loop(State, Handler) ->
    receive
        % a request from 'From' with data 'Request'
        {request, From, Ref, Request} ->
            % run handler on request
            case Handler(State, Request) of
                % get handler's output
                {reply, NewState, Result} ->
                    % the requester gets the result
                    From ! {response, Ref, Result},
                    % the server continues with the new state
                    loop(NewState, Handler)
            end;
        {stop, _From, _Ref} -> ok
    end.
```

# Generic servers: issuing a request

A generic server's function `request` takes care of sending **generic requests** to the server, and of receiving back the results:

```
% issue a request to 'Server'; return answer
request(Server, Request) ->
    Ref = make_ref(), % unique reference number
    % send request to server
    Server ! {request, self(), Ref, Request},
    % wait for response, and return it
    receive {response, Ref, Result} -> Result end.
```

# Math server: using the generic server

Here is how we can define the [math server](#) using the [generic server](#) (starting and stopping use the handling function `math_handler`):

```
start() -> gserver:start(0, fun math_handler/2).  
  
stop(Server) -> gserver:stop(Server).
```

The handling function has two cases, one per request kind:

```
math_handler(N, {factorial, M}) -> {reply, N+1, compute_factorial(M)};  
math_handler(N, status)           -> {reply, N, N}.
```

The exported functions `factorial` and `status` (called by clients) call the generic server's request function:

```
factorial(Server, M) -> gserver:request(Server, {factorial, M}).  
status(Server) -> gserver:request(Server, status).
```

# Servers: improving robustness and flexibility

We extend the implementation of the generic server to **improve**:

**robustness:** add support for error handling and crashes

**flexibility:** add support for updating the server's functionality while the server is running

**performance:** discard spurious messages sent to the server, getting rid of “junk” in the mailbox

All these extensions to the generic server do not change its **interface**

- Thus instance servers relying on it will still work, with the **added benefits** provided by the new functionality!

# Robust servers

If computing the **handling function** on the input **fails**, we **catch** the resulting exception and notify the client that an error has occurred

To handle any possible exception, use the **catch(E)** built-in function:

if evaluating **E** succeeds, the result is propagated;

if evaluating **E** fails, the resulting exception **Reason** is propagated as `{'EXIT', Reason}`

This is how we perform **exception handling** in the **event loop**:

```
case catch(Handler(State, Request)) of
  % in case of error
  {'EXIT', Reason} ->
    % the requester gets the exception
    From ! {error, Ref, Reason},
    % the server continues in the same state
    loop(State, Handler);
  % otherwise (no error): get handler's output
  {reply, NewState, Result} ->
```

# Flexible servers

Changing the server's functionality requires a new kind of **request**, which does not change the server's state but it **changes the handling function**

The event loop now receives also this new request kind:

```
% a request to swap 'NewHandler' for 'Handler'
{update, From, Ref, NewHandler} ->
  From ! {ok, Ref}, % ack
  % the server continues with the new handler
  loop(State, NewHandler);
```

Function `update` takes care of sending requests for changing handling function (similarly to what `request` does for basic requests):

```
% change 'Server's handler to 'NewHandler'
update(Server, NewHandler) ->
  Ref = make_ref(), % send update request to server
  Server ! {update, self(), Ref, NewHandler},
  receive {ok, Ref} -> ok end. % wait for ack
```

← Allows for “hot upgrading”

# Discarding junk messages

If unrecognized messages are sent to a server, they remain in the mailbox indefinitely (they never pattern match in **receive**)

If too many such “junk” messages pile up in the mailbox, they may slow down the server

To avoid this, it is sufficient to match any unknown messages and discard them as last clause in the event loop’s **receive**:

```
% discard unrecognized messages
— -> loop(State, Handler)
```

To avoid clients waiting forever for responses to discarded requests, we add a **timeout** to request:

```
receive
  {response, Ref, Result} -> Result
  % after 10 seconds, give up
  after 10000 -> timeout end.
```

# Location transparency and distribution

# Registered processes

One needs another process's pid to exchange messages with it. To increase the flexibility of [exchanging pids in open systems](#), it is possible to [register](#) processes with a symbolic name:

- `register(Name, Pid)`: register the process `Pid` under `Name`; from now on, `Name` can be used wherever a pid is required
- `unregister(Name)`: unregister the process under `Name`; when a registered process terminates, it implicitly unregisters as well
- `registered()`: list all names of registered processes
- `whereis(Name)`: return pid registered under `Name`

In the [generic server](#), we can add a registration function with name:

```
% start a server and register with 'Name'  
  
start(InitialState, Handler, Name) ->  
  
    register(Name, start(InitialState, Handler)).
```

All other server functions can be used by passing `Name` for `server`

# From concurrent to distributed

Message passing concurrency works in the same way independent of whether the processes run on the same computer or in a **distributed setting**

In Erlang, we can turn any application into a distributed one by running processes on **different nodes**:

- start an Erlang runtime environment on each node
- connect the nodes by issuing a ping
- load the modules to be execute on all nodes in the cluster
- for convenience, register the server processes
- to identify registered process `Name` running on a node `node@net-address` use the tuple `{Name, 'node@net-address'}` wherever you would normally use a registered name or pid

# Distribution: setting up nodes

In our simple experiments, the nodes are processes on the same physical local machine (IP address 127.0.0.1, a.k.a. local host), but the very same commands work on different machines connected by a network

## Node server@127.0.0.1:

```
> erl -name 'server@127.0.0.1'  
      -setcookie math-cluster  
s1>
```

## Node client@127.0.0.1:

```
> erl -name 'client@127.0.0.1'  
      -setcookie math-cluster  
c1>
```

By using the flag `setcookie` we give a symbolic name that all the nodes in a group share (only those nodes having the same cookie can interact - to avoid unwanted connections from processes in other nodes)

(A cookie is an identifier that all nodes in the same connected group share)

# Distribution: connect nodes and load modules

Nodes are invisible to each other until a message is exchanged between them; after that, they are **connected**

Node `client@127.0.0.1`:

```
% send a ping message to connect client to server node
c1> netadm:ping('server@127.0.0.1').
pong % the nodes are now connected
```

```
% list connected nodes
c2> nodes().
['server@127.0.0.1']

% load module 'ms' in all connected nodes
c3> n1(ms).
abcast % the module is now loaded
```

# Distribution: server setup

We **start the math server** on the node server and register it under the name `mserver`. Then, we can **issue request** from the client node using

`{mserver, 'server@127.0.0.1'}` instead of pids.

## Node server@127.0.0.1:

```
s1> register(mserver,ms:start()).  
true  
% server started  
% and registered
```

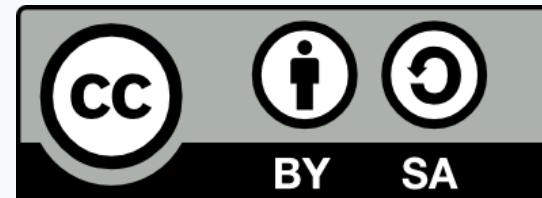
## Node client@127.0.0.1:

```
c4> ms:factorial({mserver, 'server@127.0.0.1'}, 10).  
3628800  
c5> ms:status({mserver, 'server@127.0.0.1'}).  
1  
c6> ms:status({mserver, 'server@127.0.0.1'}).  
1
```

The very same protocol works for an arbitrary number of client nodes

# These slides' licence

© 2016–2019 Carlo A. Furia, Sandro Stucki



Except where otherwise noted, this work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

To view a copy of this license, visit

<http://creativecommons.org/licenses/by-sa/4.0/>.

# Synchronization problems with message-passing

TDA384/DIT391

Principles of Concurrent Programming

Nir Piterman and Gerardo Schneider

Chalmers University of Technology | University of Gothenburg



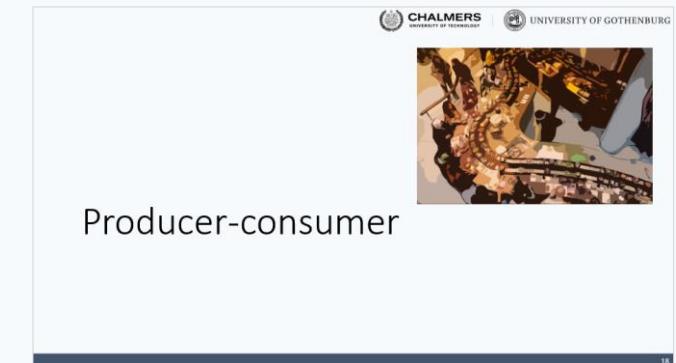
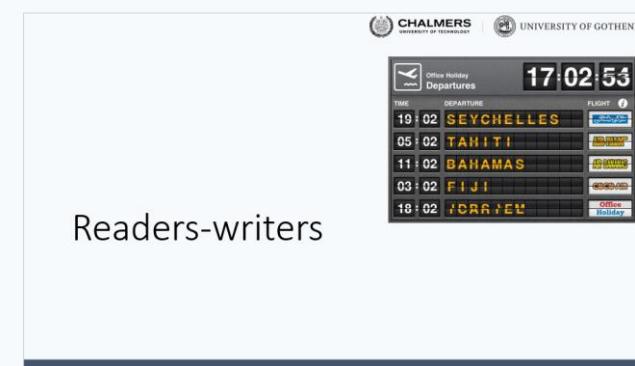
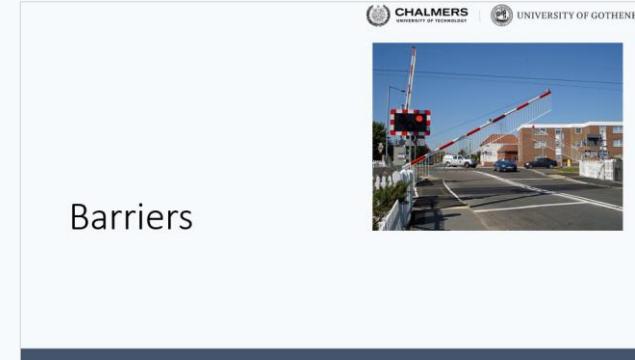
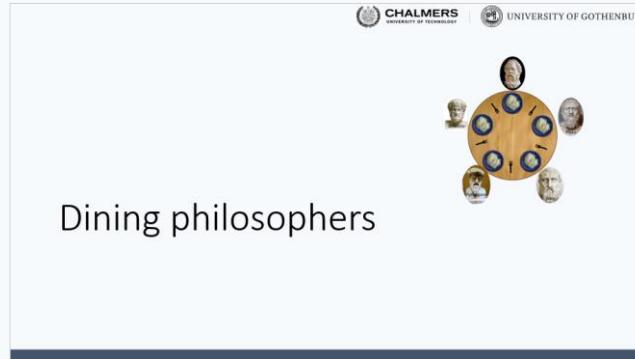
UNIVERSITY OF  
GOTHENBURG



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY

# Today's menu

- Barriers
- Resource allocator
- Producer-consumer
- Readers-writers
- Dining philosophers



# A gallery of synchronization problems

In today's class, we go through several **classical synchronization problems** and solve them using processes and **message passing**

On the course website you can download fully working implementations of some of the problems

Solving these problems with message passing has a **different style** than using semaphores or monitors:

- **Mutual exclusion** is not an issue, since there are **no shared variables**
- **Coordination** is the main problem, which is achieved by exchanging messages asynchronously

The solutions are in the style of **servers**, which run event-loop functions that handle requests from clients thus **coordinating** them

# Barriers



# Reusable barriers – recap

-module(barrier).

% initialize barrier for ‘Expected’ processes

init(Expected) -> todo.

% block at ‘Barrier’ until all processes have reached it

wait(Barrier) -> todo.

**Reusable barrier:** implement module barrier such that:

- A process blocks on wait until all processes have reached the Barrier
- After Expected threads have executed wait, the barrier is closed again

# Processes at a reusable barrier

Processes **continuously approach** the barrier, which must guarantee that they synchronize each access.

---

process<sub>k</sub>

```
process(Barrier) ->
    % code before barrier
    barrier:wait(Barrier) % synchronize at barrier
    % code after barrier
process(Barrier).
```

# Barrier process

The **barrier** process keeps track of the processes that have arrived at the barrier:

- when a new process **arrives**, it sends an arrived message to the barrier; the barrier updates its list of arrived processes
- when the list of arrived processes is **complete**, the barrier sends a continue message to all processes
- after notifying all processes, the barrier goes back to the **initial state**, ready for a new iteration

We implement the barrier's event loop as a **server function**:

```
barrier(Arrived, Expected, PidRefs)
```

where **Arrived** processes have arrived so far, out of a total of **Expected**; **PidRefs** is a list of the pids and unique references of **arrived** messages sent to the barrier (thus it has **Arrived** elements)

# The server function barrier

```

% event loop of barrier for 'Expected' processes
%   Arrived: number of processes arrived so far
%   PidRefs: list of {Pid, Ref} of processes arrived so far
barrier(Arrived, Expected, PidRefs) when Arrived =:= Expected -> % all processes arrived
  % notify all waiting processes:
  [To ! {continue, Ref} || {To, Ref} <- PidRefs],
% reset barrier:
barrier(0, Expected, []);
barrier(Arrived, Expected, PidRefs) ->
  receive % still waiting for some processes
    {arrived, From, Ref} ->
      % one more arrived: add {From, Ref} to PidRefs list:
      barrier(Arrived+1, Expected, [{From, Ref}|PidRefs])
  end.

```

List comprehension: Go through the list of all pairs of PidRefs, extract each component of the pair into To (process PId) and Ref (instance of the process arriving to barrier) and send a message to that particular instance with the message continue

Arrived is redundant because it is equal to `length(PidRefs)`; we keep it for clarity

# The function wait

The function `wait` exchanges messages with the `Barrier` process running barrier; it is used so that synchronizing processes do not need to know about the format of exchanged messages

```
% block at 'Barrier' until all processes have reached it
wait(Barrier) ->
    Ref = make-ref(),
    % notify barrier of arrival
    Barrier ! {arrived, self(), Ref},
    % wait for signal to continue
    receive {continue, Ref} -> through end.
```

↑  
dummy value

# Barrier initialization

Initializing a barrier consists of spawning a process running barrier

```
% initialize barrier for 'Expected' processes
init(Expected) ->
    spawn(fun () -> barrier(0, Expected, []) end).
```

↑      →  
initially, no processes have arrived yet

The caller gets the barrier's pid, which should be distributed to all processes that want to use the barrier

# Resource allocator

# Resource allocator: the problem – recap

An **allocator** grants **users** exclusive access to a number of resources:

- **users** asynchronously request resources and release them back
- the **allocator** ensures resources are given exclusively to one user at a time, and keeps tracks of how many resources are available

```
-module(allocator).  
% register 'allocator' with list of Resources  
init(Resources) -> todo.  
% get 'N' resources from 'allocator'  
request(N) -> todo.  
% release 'Resources' to 'allocator'  
release(Resources) -> todo.
```

**Resource allocator** problem: implement allocator such that:

- an arbitrary number of users can access the allocator
- users are granted exclusive access to resources

# Users

Users continuously and asynchronously access the allocator, which must guarantee proper synchronization

---

user<sub>k</sub>

```
user() ->
    % how many resources are needed?
    N = howMany(),
    % get resources from allocator
    Resources = allocator:request(N),
    % do something with resources
    use(Resources),
    % release resources
    allocator:release(Resources),
    user().
```

# Allocator process

The **allocator** process keeps track of the list of available resources:

- when a process **requests** some resources that are available, the allocator sends a granted message to the process, and removes those just granted from the list of available resources
- when a process **releases** some resources, the allocator sends a released message to the process, and adds those just released to the list of available resources
- requests that **exceed** the availability implicitly queue in the allocator's mailbox; they will be served as soon as enough resources are available

We implement the allocator's event loop as a server function:

```
allocator(Resources)
```

where **Resources** is the list of available resources

# The server function allocator: handling requests

```
allocator(Resources) ->
    % count how many resources are available
    Available = length(Resources),
    receive
        % serve requests if enough resources are available
        {request, From, Ref, N} when N <= Available ->
            % Granted ++ Remaining := Resources
            % length(Granted) := N
            {Granted, Remaining} = lists:split(N, Resources),
            % send resources to requesting process
            From ! {granted, Ref, Granted},
            % continue with Remaining resources
            allocator(Remaining);
```

[Continue in next slide...]

# The server function allocator: handling releases

```
allocator(Resources) ->
  % count how many resources are available
  Available = length(Resources),
receive
  % serve requests: previous slide...
  % serve releases
  {release, From, Ref, Released} ->
    % notify releasing process
    From ! {released, Ref},
    % continue with previous and released resources
    allocator(Resources ++ Released)
end.
```

# The functions request and release

The functions `request` and `release` exchange messages with the process registered as allocator; they are used so that synchronizing processes do not need to know about the format of exchanged messages

```
% get 'N' resources from 'allocator'; block if not available
request(N) ->
  Ref = make_ref(),
  allocator ! {request, self(), Ref, N},
  receive {granted, Ref, Granted} -> Granted end.
```

```
% release 'Resources' to 'allocator'
release(Resources) ->
  Ref = make_ref(),
  allocator ! {release, self(), Ref, Resources},
  receive {released, Ref} -> released end.
```



# Producer-consumer

# Producer-consumer: the problem – recap

```
-module(buffer).  
% initialize buffer with size 'Bound'  
init_buffer(Bound) -> todo.  
% put 'Item' in 'Buffer'; block if full  
put(Buffer, Item) -> todo.  
% get item from 'Buffer'; block if empty  
get(Buffer) -> todo.
```

Producer-consumer problem: implement buffer such that:

- producers and consumer access the buffer atomically
- consumers block when the buffer is empty
- producers block when the buffer is full (bounded buffer variant)

# Producers and consumers

Producers and consumers continuously and asynchronously access the buffer, which must guarantee proper synchronization

---

producer<sub>n</sub>

```
producer(Buffer) ->
    % create a new item
    Item = produce(),
    buffer:put(Buffer, Item),
    producer(Buffer).
```

consumer<sub>m</sub>

```
consumer(Buffer) ->
    Item = buffer:get(Buffer),
    % do something with 'item'
    consume(Item),
    consumer(Buffer).
```

Note that **atomic access** is not an issue with processes: a single sequential process will actively modify the content of the buffer in response to messages sent by other processes

# Buffer process: bounded buffer

The **buffer** process keeps track of the items stored in the buffer:

- when a process asks to **get** one item and the **buffer is not empty**, the buffer sends an item message to the process, and removes the item just taken from the buffer list
- when a process asks to **put** one item and the **buffer is not full**, the buffer sends a done message to the process, and adds the item just sent to the buffer list
- as in the allocator example, requests that cannot be satisfied (get with empty buffer, and put with full buffer) implicitly queue in the allocator's mailbox; they will be served as soon as it is possible

We implement the buffer's event loop as a server function:

```
buffer(Content, Count, Bound)
```

where **Content** is the list of **Count** available resources and **Bound** is the buffer's size

# The server function buffer: handling requests

```
buffer(Content, Count, Bound) ->  
receive  
    % serve gets when buffer not empty  
    {get, From, Ref} when Count > 0 ->  
        [First|Rest] = Content,      % match first item  
        From ! {item, Ref, First}, % send it out  
        buffer(Rest, Count-1, Bound); % remove it from buffer  
    % serve puts when buffer not full  
    {put, From, Ref, Item} when Count < Bound ->  
        From ! {done, Ref},           % send ack  
        buffer(Content ++ [Item], Count+1, Bound) % add item to end
```

**end.**

Starvation **not** possible: when buffer is neither full nor empty, requests are served in the order they arrive

If buffer fills up, **put** is disabled; after finitely many **gets** are served, buffer no longer full, which disables **get**, thus allowing **put** to be served

Similarly, **put** activates **get** when the buffer is empty

# Buffer process: unbounded buffer

In an **unbounded buffer**, the condition `Count < Bound` always holds:

```
% serve puts
{put, From, Ref, Item} when Count < Bound ->
% ...
```

Instead of removing the condition (as well as all the occurrences of `Bound`), we can take advantage of Erlang's order between numbers and atoms (every number is less than any atom): setting `Bound` to infinity ensures that `Count < Bound` will always evaluate to true

This way, we can use the very same implementation both in the bounded and in the unbounded case

# The functions `get` and `put`

The functions `get` and `put` exchange messages with the process with pid `Buffer`; they are used so that synchronizing processes do not need to know about the format of exchanged messages

```
% get item from 'Buffer'; block if empty
get(Buffer) ->
    Ref = make_ref(),
    Buffer ! {get, self(), Ref},
    receive {item, Ref, Item} -> Item end.
```

```
% put 'Item' in 'Buffer'; block if full
put(Buffer, Item) ->
    Ref = make_ref(),
    Buffer ! {put, self(), Ref, Item},
    receive {done, Ref} -> done end.
```



# Readers-writers

# Readers-writers: the problem – recap

```
-module(board).  
  
init(Name) -> todo.          % register board with 'Name'  
begin_read(Board) -> todo.    % get read access to 'Board'  
end_read(Board) -> todo.      % release read access to 'Board'  
begin_write(Board) -> todo.   % get write access to 'Board'  
end_write(Board) -> todo.     % release write access to 'Board'
```

**Readers-writers** problem: implement board such that:

- multiple reader can operate concurrently
- each writer has exclusive access

Invariant:  $\#Writers = 0 \vee (\#Writers = 1 \wedge \#Readers = 0)$

Other properties that a good solution should have:

- support an arbitrary number of readers and writers
- no starvation of readers or writers

# Readers and writers

Readers and writers continuously and asynchronously try to access the board, which must guarantee proper synchronization

---

reader<sub>n</sub>

```
reader(Board) ->  
    board:begin-read(Board),  
    % read messages  
    board:end-read(Board),  
    reader(Board).
```

writer<sub>m</sub>

```
writer(Board) ->  
    board:begin-write(Board),  
    % write messages  
    board:end-write(Board),  
    writer(Board).
```

# Board process – first version

A first solution to the readers-writers problem can **extend the idea behind the allocator**: serve requests when possible and let other requests queue in the mailbox

The board process keeps track of number of readers and writers active on the board:

- when a new request to **begin reading** arrives and no writer is active, the board sends an OK to read message to the requester, and increases the count of readers;
- when a new request to **begin writing** arrives and no readers or writers are active, the board sends an OK to write message to the requester, and increases the count of writers;
- conversely, when notifications to **end read** or **end write** arrive, the board decreases the count of readers or writers;
- requests that **cannot be served implicitly** queue in the board's mailbox; they will be served as soon as the board is freed

# The server function board-Row – first version

```
% 'Readers' active readers and 'Writers' active writers
board_Row(Readers, writers) ->
receive
    {begin_read, From, Ref} when writers =:= 0 ->
        From ! {ok_to_read, Ref},
        board_Row(Readers+1, writers);
    {begin_write, From, Ref} when (writers =:= 0) and (Readers =:= 0) ->
        From ! {ok_to_write, Ref},
        board_Row(Readers, writers+1);
    {end_read, From, Ref} -> From ! {ok, Ref},
        board_Row(Readers-1, writers);
    {end_write, From, Ref} -> From ! {ok, Ref},
        board_Row(Readers, writers-1)
end.
```

# Readers-writers: the first version prioritizes readers

In board-Row, the “waiting conditions” follow directly from the invariant; thus, the solution is **correct** in that it ensures **mutual exclusion** according to the readers-writers invariant

However, it gives **priority** to readers over writers:

- new reading requests get served without waiting as long as a reader is active
- writing requests waiting in the mailbox have to wait until the last reader sends an end-read message
- as long as reading requests keep arriving and queuing in the mailbox, the waiting writing requests will never execute

**Exchanging the order** of clauses in the **receive** does not solve the problem (nor does it give priority to writers over readers): readers can still starve writers because the condition for writing is **stronger** than the condition for reading, and writers cannot maintain their condition without the cooperation of readers

# Readers-writers: towards a fair solution

We could achieve **fairness** by replicating the pattern behind the solution with monitors

- the board keeps track of the lists of **pending** read and write **requests**
- **read requests** are served as long as there are no active writers and no pending write requests
- notifications to end-write let in one pending read request, or one waiting write request if there are no reading requests

This approach **works**, but it is quite **cumbersome** to implement with message passing

Main issue: it requires a duplication of the information that is already implicit in the mailbox queue, which complicates ensuring that messages are processed exactly once

# Readers-writers: fair solution

We implement a fair solution where the board can be in one of two macro states:

**empty**: there are neither active readers nor active writers

**readers**: there are some active readers and no active writers

When the board is in macro state **empty**:

- **read requests** are served immediately, then the board switches to macro state **readers**
- **write requests** are served immediately and synchronously: the board waits until writing ends, then the board is **empty** again

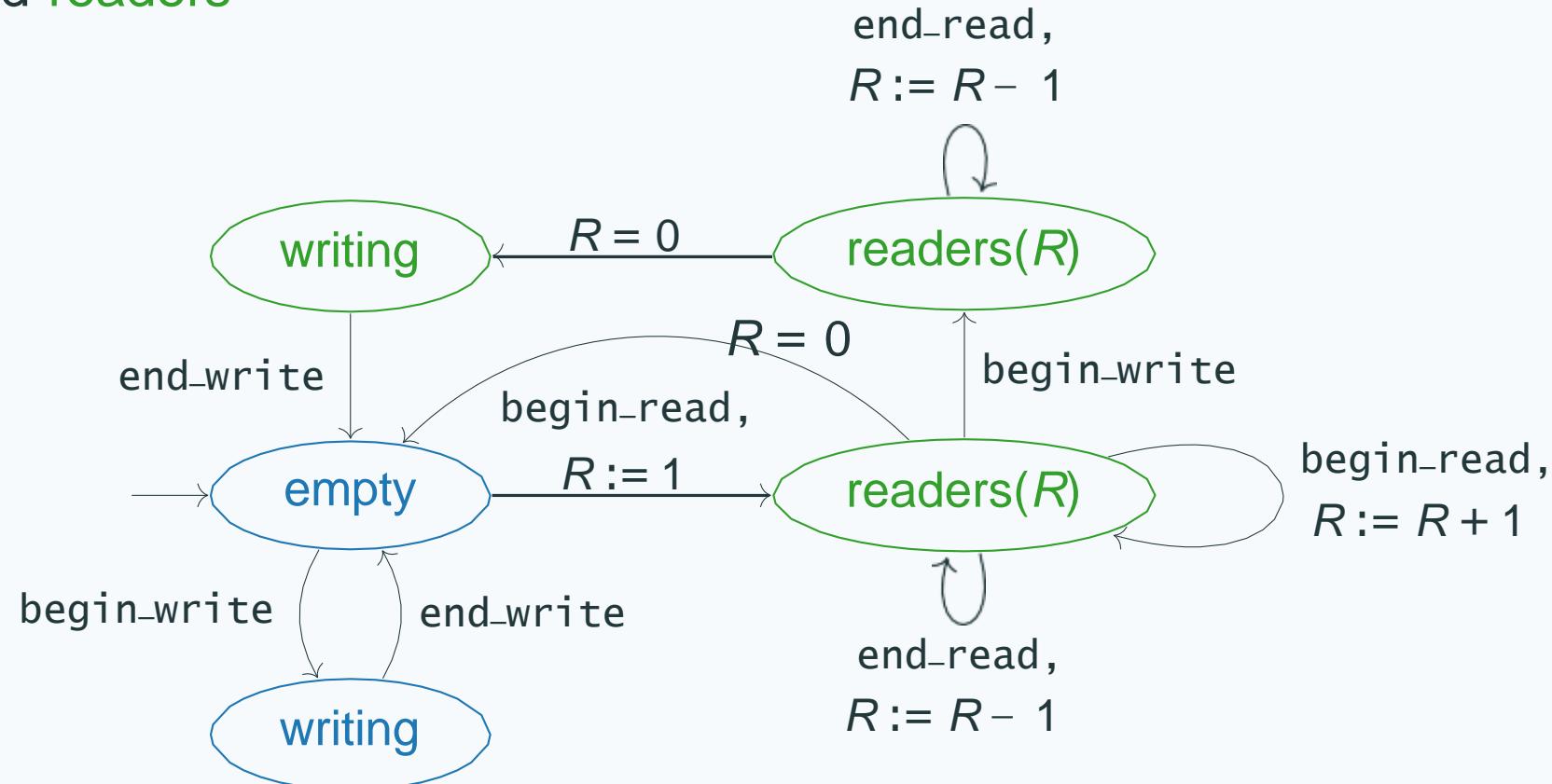
When the board is in macro state **readers**:

- **read requests** are served immediately, and the macro state remains **readers**
- **write requests** are served as soon as possible: the board waits until all reading ends, then the writing request is served synchronously, and then the board is **empty** again

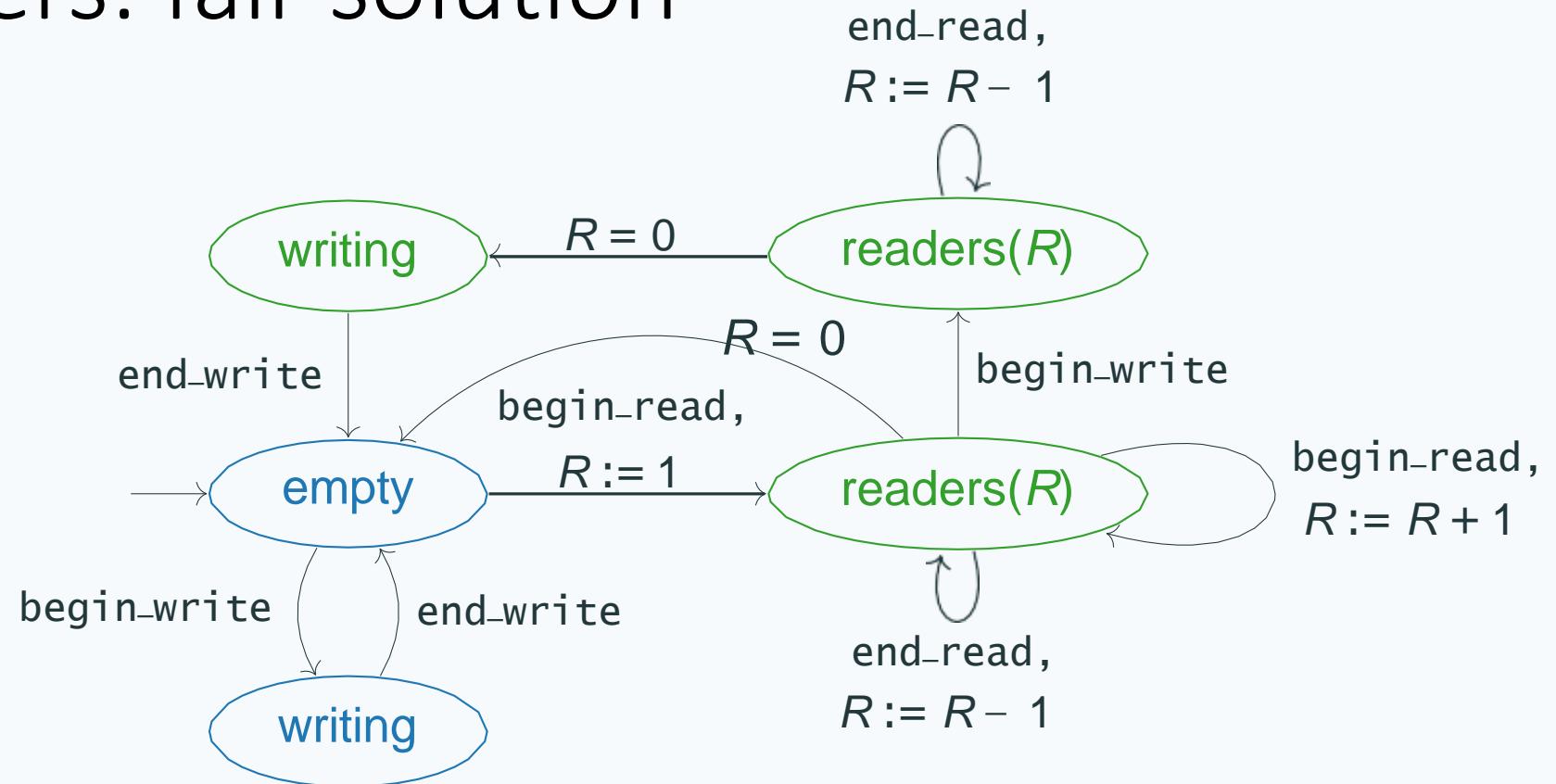
# Readers-writers: fair solution (cont'd)

This state/transition diagram formalizes the solution illustrated informally above

The partitioning of states in the diagram according to their color corresponds to the macro states **empty** and **readers**



# Readers-writers: fair solution (cont'd)



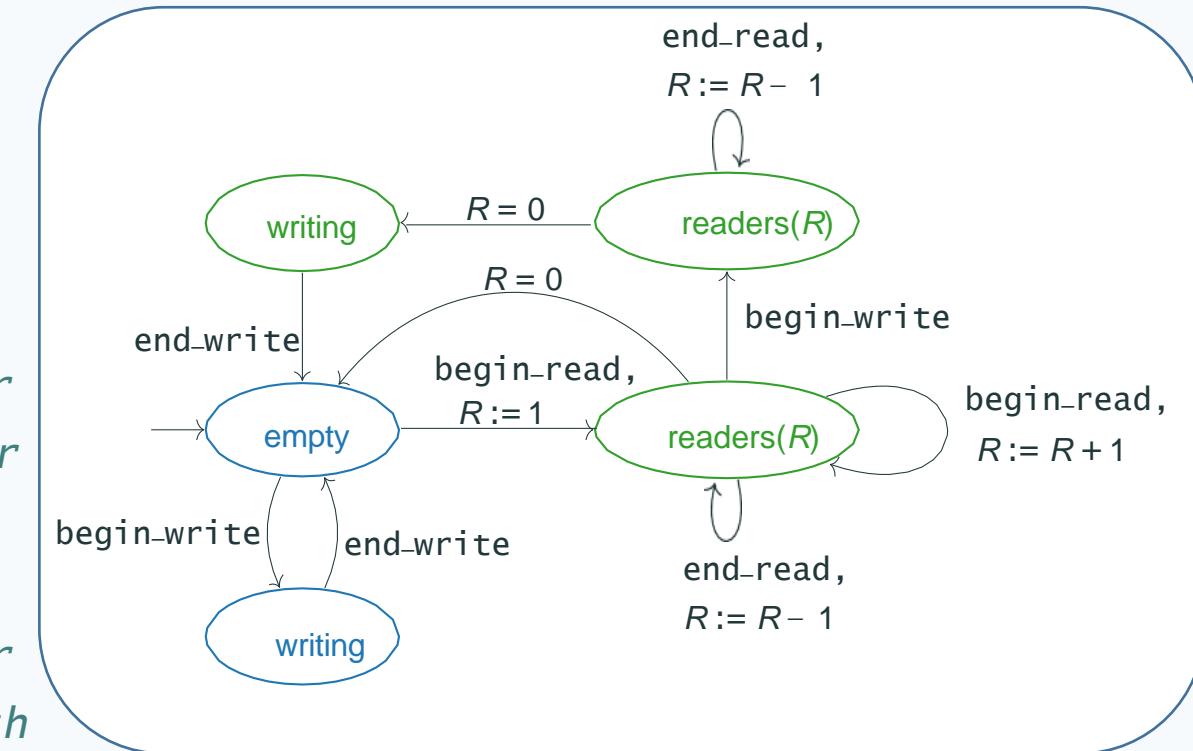
By inspecting the diagram: it guarantees fairness **provided** outgoing transitions from the same state have the same priority (they are served in arrival order)

The solution in Erlang implements the behavior of this diagram, using **two** server functions `empty_board` and `readers_board`, which call each other

# The server function empty-board

```
% board with no readers and no writers
empty_board() ->

  receive
    % serve read request
    {begin-read, From, Ref} ->
      From ! {ok-to-read, Ref}, % notify reader
      readers_board(1); % board has one reader
    % serve write request synchronously
    {begin-write, From, Ref} ->
      From ! {ok-to-write, Ref}, % notify writer
      Receive % wait for writer to finish
        {end-write, _From, _Ref} ->
          empty_board() % board is empty again
      end
  end.
```



# The server function readers-board: serving write requests

```
% board with no readers (and no writers)
readers-board(0) -> empty_board();
```

```
% board with 'Readers' active readers
% (and no writers)
readers-board(Readers) ->
```

**receive**

% serve write request

```
{begin-write, From, Ref} ->
```

% wait until all 'Readers' have finished

```
[receive {end-read, _From, _Ref} -> end-read end || - <- lists:seq(1, Readers)],
```

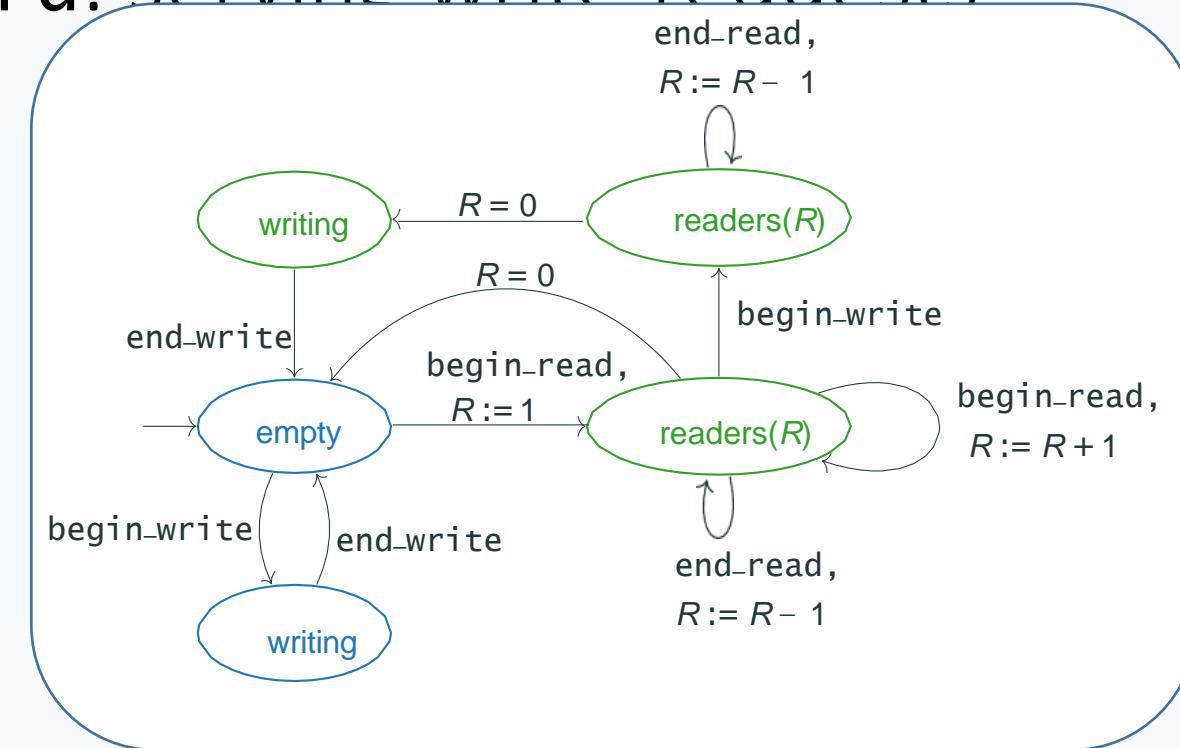
From ! {ok-to-write, Ref}, % notify writer

**receive** % wait for writer to finish

```
{end-write, _From, _Ref} -> empty_board()
```

**end:** % board is empty again

[Continue in next slide...]



Take all active readers and wait till all finiish and send end-read to all (one by one)

# The server function `readers-board`: serving read requests

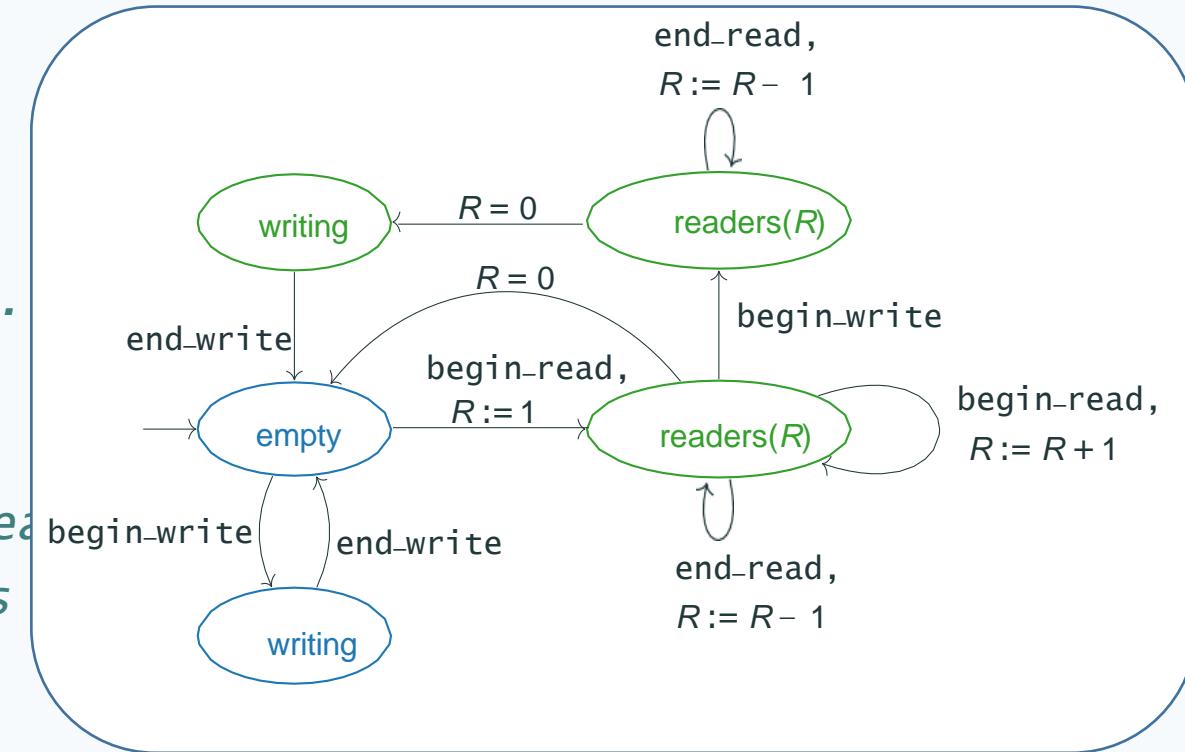
Now the order of clauses in the `receive` does not matter: requests are processed in the mailbox order because none of the three clauses (begin-read, end-read, and begin-write) has a condition stronger than the others

```
readers-board(Readers) ->
  receive
    % serve write requests: previous slide...
    % serve read request
    {begin-read, From, Ref} ->
      From ! {ok_to_read, Ref}, % notify reader
      readers-board(Readers+1); % board has one more reader
    % serve end read
    {end-read, _From, _Ref} ->
      readers-board(Readers-1) % board has one less reader
  end.
```

# The server function `readers-board`: serving read requests

Now the order of clauses in the `receive` does not matter: requests are processed in the mailbox order because none of the three clauses (`begin-read`, `end-read`, and `begin-write`) has a condition stronger than the others

```
readers-board(Readers) ->
  receive
    % serve write requests: previous slide..
    % serve read request
    {begin-read, From, Ref} ->
      From ! {ok_to_read, Ref}, % notify reader
      readers-board(Readers+1); % board has one more reader
    % serve end read
    {end-read, _From, _Ref} ->
      readers-board(Readers-1) % board has one less reader
  end.
```



## The functions `begin-read`, `end-read`, `begin-write`, and `end-write`

The functions `begin-read`, `end-read`, `begin-write`, and `end-write` exchange messages with the board server process with pid `Board`; they are used so synchronizing processes don't need to know about the format of exchanged messages

For example:

```
% get read access to 'Board'  
begin-read(Board) ->  
  Ref = make-ref(),  
  Board ! {begin-read, self(), Ref},  
  receive  
    {ok_to_read, Ref} -> ok_to_read  
end.
```

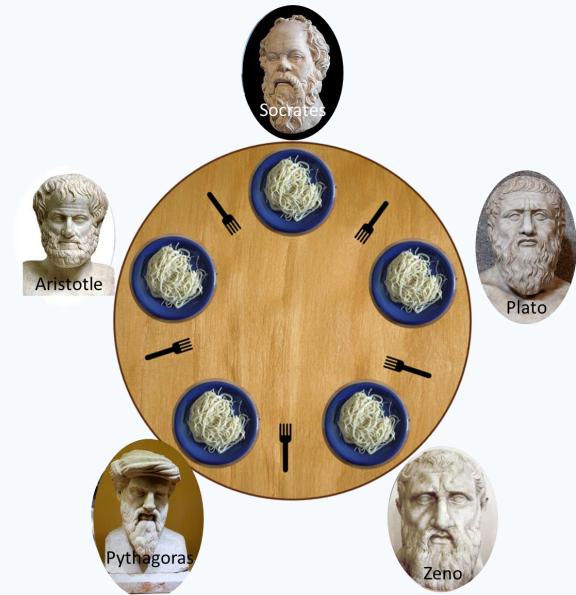
The behavior of the board process changes over time, but the pid `Board` stays the same

# Board initialization

Initializing a board consists of spawning a process running `empty-board`.

```
% initialize empty board and register with 'Name'  
init(Name) ->  
    register(Name, spawn(fun empty-board/0)).
```

After initialization, `Name` can be used to access the board



# Dining philosophers

# Dining philosophers: the problem – recap

-module(philosophers).

% set up table of 'N' philosophers

init(N) -> todo.

% philosopher picks up 'Fork'

get-fork(Fork) -> todo.

% philosopher releases 'Fork'

put-fork(Fork) -> todo.



**Dining philosophers** problem: implement philosophers such that:

- forks are held exclusively by one philosopher at a time
- each philosopher only accesses adjacent forks
- no philosopher starves

# Philosophers with waiter

We could replicate solutions based on locking; e.g. setting up a server for each **pair of forks**, which grants access to both forks atomically to the first philosopher that sends a request

Instead, let's explore an approach that is more **congenial to message passing**

A **waiter process** supervises access to the table

Each philosopher asks the waiter for **permission to sit** before picking up both forks and notifies the waiter after putting down both forks

As long as the waiter allows **strictly fewer** philosophers than the total number of forks to sit around the table at the same time, **deadlock** and **starvation** are avoided

The **waiter's interface** consists of two functions:

```
% ask 'Waiter' to be seated; may wait
sit(waiter) -> todo.
% ask 'Waiter' to leave
leave(waiter) -> todo.
```

# Philosophers

Philosophers continuously alternate between thinking and eating, while coordinating with the waiter

---

```
philosopherk

% Forks: fork#{left, right} of fork pids
% Waiter: waiter process
philosopher(Forks, Waiter) -> think(),
    sit(waiter),                                % ask to be seated
    get-fork(Forks#forks.left),                  % pick up left fork
    get-fork(Forks#forks.right),                 % pick up right fork
    eat(),
    put-fork(Forks#forks.left),                  % put down left fork
    put-fork(Forks#forks.right),                 % put down right fork
    leave(waiter),                            % notify leaving
philosopher(Forks, Waiter).
```

# Waiter process

The **waiter** process keeps track of how many philosophers are eating at the table:

- when a philosopher asks to be **seated** and table is not full, waiter sends an `ok-to-sit` message to the philosopher and increases the count of eating philosophers
- when a philosopher notifies **leaving**, waiter sends an `ok-to-leave` message to the philosopher and decreases the count of eating philosophers
- requests to sit that arrive when the table is full queue in the waiter's mailbox; they will be served as soon as a seat frees up

We implement the waiter's event loop as a server function:

```
waiter(Eating, Seats)
```

where `Eating` philosophers are sitting and eating, out of a total of `Seats` available seats (`Seats` is the number of seats that can be occupied at the same time)

# The server function waiter

```

waiter(Eating, Seats) ->
receive
  % serve as long as seats are available
  {sit, From, Ref} when Eating < Seats ->
    io:format("~p eating (~p at table)~n", [From, Eating+1]),
    From ! {ok_to_sit, Ref},
    waiter(Eating+1, Seats);           % one more eating
  % can leave at any time
  {leave, From, Ref} ->
    io:format("~p leaving (~p at table)~n", [From, Eating-1]),
    From ! {ok_to_leave, Ref},
    waiter(Eating-1, Seats)           % one less eating
end.

```

(Printing the table's state at every change is for debugging purposes)

# The functions sit and leave

Two handler functions: sit and leave (they hide the format of messages exchanged between waiter and philosophers)

```
% ask 'Waiter' to be seated; may wait
sit(Waiter) ->
    Ref = make_ref(),
    Waiter ! {sit, self(), Ref},
    receive {ok-to-sit, Ref} -> ok end.
```

```
% ask 'Waiter' to leave
leave(Waiter) ->
    Ref = make_ref(),
    Waiter ! {leave, self(), Ref},
    receive {ok-to-leave, Ref} -> ok end.
```

# The fork processes and functions

Each fork has a `fork` process which keeps track of whether the fork is free (on the table) or held by a philosopher

The server function for a fork can be in two states (whether the fork is held or not)

```
% a fork not held by anyone
fork() ->
    receive
        {get, From, Ref} ->
            From ! {ack, Ref},
            fork(From) % fork held
    end.
```

```
% a fork held by Owner
fork(Owner) ->
    receive
        {put, Owner, _Ref} ->
            fork() % fork not held
    end.
```

For simplicity, `put` requests don't get an acknowledgment; they take effect immediately

# The functions get-fork and put-fork

The structure of get-fork and put-fork are similar to things we've seen:

```
% pick up 'Fork'; block until available
get_fork(Fork) ->
    Ref = make_ref(),
    Fork ! {get, self(), Ref},
    receive {ack, Ref} -> ack end.
```

```
% put down 'Fork'
put_fork(Fork) ->
    Ref = make_ref(),
    Fork ! {put, self(), Ref}.
```

# Table initialization

Initializing a table consists of spawning the processes running waiter, fork and philosopher, as well as connecting each philosopher to their pair of forks

```
% set up table of 'N' philosophers
init(N) ->
    % spawn waiter process
    waiter = spawn(fun () -> waiter(0, N-1) end),
    Ids = lists:seq(1,N), % [1, 2, ..., N]
    % spawn fork processes
    Forks = [spawn(fun fork/0) || _ <- Ids],
    % spawn philosopher processes
    [spawn(fun () ->
        Left = lists:nth(I, Forks),
        Right = lists:nth(1+(I rem N), Forks), % 1-based indexes
        philosopher(#forks{left=Left, right=Right}, waiter)
    end) || I <- Ids].
```

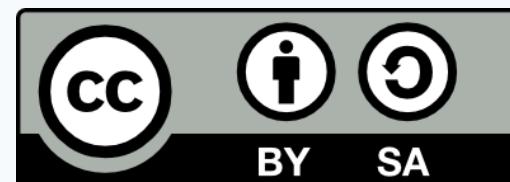
at most  $N-1$  eating philosophers at once

Different from how we numbered philosophers and forks in previous lecture: we start from 1 instead of 0, so the forks are also numbered 1..N

First get each one of the Ids from the list Ids, and spawn a corresponding fork for that ID

# These slides' license

© 2016–2019 Carlo A. Furia, Sandro Stucki



Except where otherwise noted, this work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

To view a copy of this license, visit

<http://creativecommons.org/licenses/by-sa/4.0/>.

# Parallel Linked Lists (sets)

Lecture 10 of TDA384/DIT391

Principles of Concurrent Programming

Nir Piterman and Gerardo Schneider

Chalmers University of Technology | University of Gothenburg



UNIVERSITY OF  
GOTHENBURG



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY

# Synchronization costs

A number of factors **challenge** designing correct and efficient **parallelizations**:

- sequential dependencies
- synchronization costs
- spawning costs
- error proneness and composability

In this lecture, we focus on reducing the **synchronization costs** associated with **locking**

# Today's menu

The burden of locking

Linked set implementations

Nodes, lists, and sets

Sequential access

Parallel linked sets

Coarse-grained locking

Fine-grained locking

Optimistic locking

Lazy node removal

Lock-free access

# The burden of locking

# The trouble with locks

Standard techniques for concurrent programming are ultimately based on **locks**

Programming with locks has several **drawbacks**:

- Performance overhead
- Lock granularity is hard to choose:
  - not enough locking: race conditions
  - too much locking: not enough parallelism
- Risk of deadlock and starvation
- Lock-based implementations do not compose
- Lock-based programs are hard to Maintain and modify

Message-passing programming is higher-level, but it also inevitably incurs on synchronization costs – of magnitude comparable to those associated with locks

# Breaking free of locks

Lock-free programming takes a fresh look at the problems of concurrency and tries to dispense with using locks altogether

- Lock-based programming is pessimistic: be prepared for the worst possible conditions:

if things can go wrong, they will

- Lock-free programming is optimistic: do what you have to do without worrying about race conditions:

if things go wrong, just try again

# Lock-free programming

Lock-free programming relies on:

- using **stronger primitives** for atomic access
- building **optimistic algorithms** using those primitives

Compare-and-set operations are an example of stronger primitives:

```
public class AtomicInteger {  
    // atomically set to 'update' if current value is 'expect'  
    // otherwise do not change value and return false  
    boolean compareAndSet(int expect, int update)  
}
```

- **Test-and-set**: modifies the contents of a memory location and returns its old value as a single atomic operation
- **Compare-and-set**: atomically compares the contents of a memory location to a given value and, *only if they are the same*, modifies the contents of that memory location to a given new value



To update an **AtomicInteger** variable k:

```
do { // keep trying until no one changes k in between  
    int oldvalue = k.get();  
    int newValue = compute(oldvalue);  
} while (!k.compareAndSet(oldvalue, newValue));
```

# Compare-and-set is not free

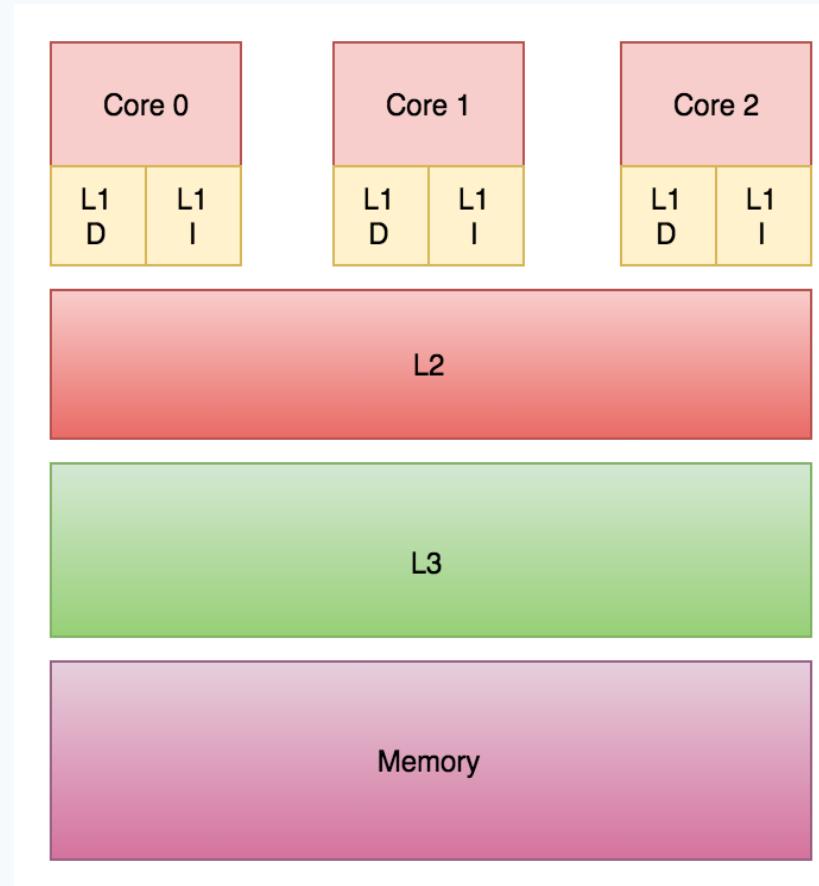


Diagram by Avadlam3, [Wikipedia \(2016\)](#).

You need to add synchronization caches to ensure memory consistency (which takes between 100 and 1000 cycles)

CAS operations are **not free**: they involve memory barrier operations to synchronize caches (~100-1000 cycles)

# Compare-and-set is not free

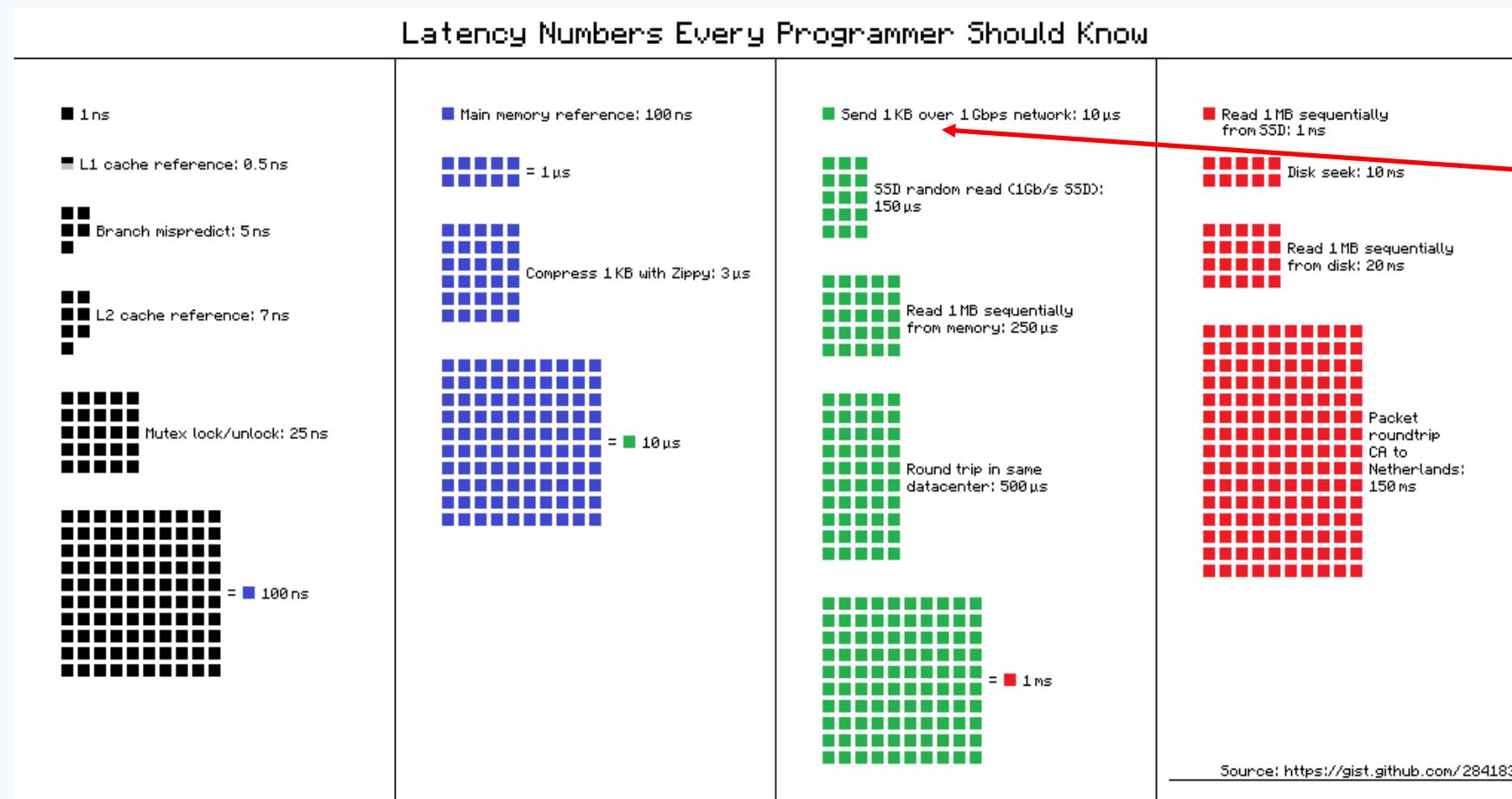


Chart by [ayshen](#), based on Peter Norvig's "[Teach Yourself Programming in Ten Years](#)".

CAS operations are **not free**: they involve memory barrier operations [to synchronize caches](#) (~100-1000 cycles)

# Lock-free vs. wait-free

Two **classes** of **lock-free algorithms**, collectively called **non-blocking**:

- **lock-free**: guarantee system-wide progress: infinitely often, some process makes progress
- **wait-free**: guarantee per-process progress: every process eventually makes progress

Which one is **stronger**?

Wait-free is **stronger** than lock-free:

- **Lock-free** algorithms are free from **deadlock**
- **Wait-free** algorithms are free from **deadlock and starvation**

# Thread-safe data structures

Programming correctly without using locks is challenging

Instead of trying to develop general techniques, we focus on implementing **reusable data structures** that make minimal usage of locking

The effort involved in developing correct implementations pays off since very many applications can then use such **thread-safe data structure** implementations to synchronize **safely** and **implicitly** by accessing the structures through their APIs

A data structure is **thread safe** if its operations are **free from race conditions** when executed by multi-threaded clients

Our **lock-free** and **wait-free** algorithms are some of those used in the implementations of **thread-safe** structures in **java.util.concurrent** (non-blocking data structures atomically accessible in parallel)

**Race condition:** the correctness of the program depends on the execution

# Linked set implementations

# Parallel linked lists

In the rest of this lecture, we go through several implementations of **linked lists** that support **parallel access**; the implementations differ in how much locking they use to guarantee correctness and, correspondingly, in how much parallelism they allow

We will use **pseudo-code** that is very close to regular Java syntax but occasionally takes some liberties to simplify the notation

On the course website you can download fully working implementations of some of the classes

# Linked set implementations

## Nodes, Lists, and Sets

# The interface of a set

We use linked lists to implement a **set** data structure with interface:

```
public interface Set<T>
{
    // add 'item' to set; return false if 'item' is already in the set
    boolean add(T item);

    // remove 'item' from set; return false if 'item' not in the set
    boolean remove(T item);

    // is 'item' in set?
    boolean has(T item);
}
```

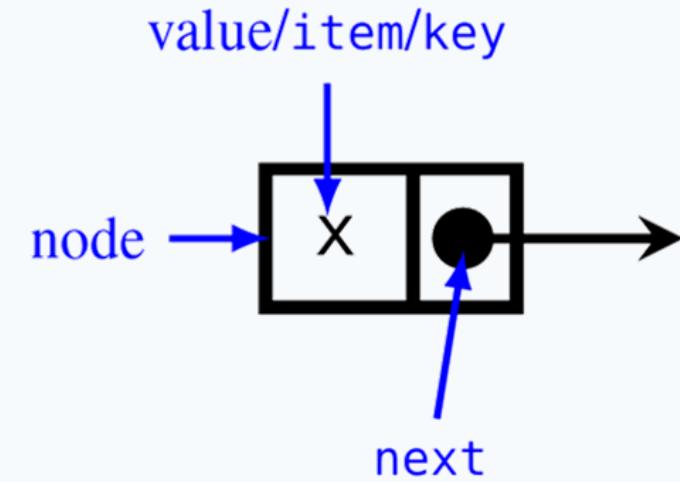
# Nodes

The underlying implementations of sets use **singly-linked lists**, which are made of chains of nodes - Every **node**:

- stores an **item** – its **value**
- has a unique **key** – the value's **hash code**
- points to the **next** node in the chain

In the graphical representations of nodes, we do not distinguish between items and their keys – and represent both by characters:

```
interface Node<T>
{
    // value of node
    T item();
    // hash code of value
    int key();
    // next node in chain
    Node<T> next();
}
```



# Lists as chains of nodes

A list with special **head** and **tail** nodes implements a **set**:

- the **elements** of the set are items in different nodes
- to facilitate searching, the nodes are maintained **sorted** in ascending **key** order
- to facilitate searching, the head has the smallest possible key, the tail has the largest possible key, and all elements have **finitely many** keys that are in between

For example, the set **{b, e, a, f, g}** is implemented by:



Relaxing these assumptions is possible at the cost of complicating the implementations

# Linked set implementations

## Sequential access

# Sequential set: basic linked implementation

We start with a standard linked-list-based implementation of sets, which **only** works for sequential access

```
class SequentialSet<T> implements Set<T>
{
    // nodes at beginning and end
    protected Node<T> head, tail;

    // empty set
    public SequentialSet() {
        head = new SequentialNode<>(Integer.MIN_VALUE);
        tail = new SequentialNode<>(Integer.MAX_VALUE);
        head.setNext(tail);
    }
}
```

Only visible within the class,  
not from any other class  
(including subclasses)

In Java:  $-2^{31}$

In Java:  $2^{31} - 1$

Empty set: head  tail

# Nodes in a sequential set

A node's implementation uses **private attributes** with **getters** and **setters**

A bit tedious (we could just let the set implementations access the attributes directly)...  
... but it leads to nicer designs in the variants of set implementations we describe later

```
class SequentialNode<T> implements Node<T> {
    private T item;          // value stored in node
    private int key;         // hash code of item
    private Node<T> next;   // next node in chain

    // getters:                                // setters:
    T item() { return item; }
    int key() { return key; }
    Node<T> next() { return next; }

    void setItem(T item) { this.item = item; }
    void setKey(int key) { this.key = key; }
    void setNext(Node<T> next) { this.next = next; }
}
```

# Finding a position inside a list

Since we maintain nodes in order of key, and every item has a unique key, we can **search** for the position of any given key by going through the list from head to tail

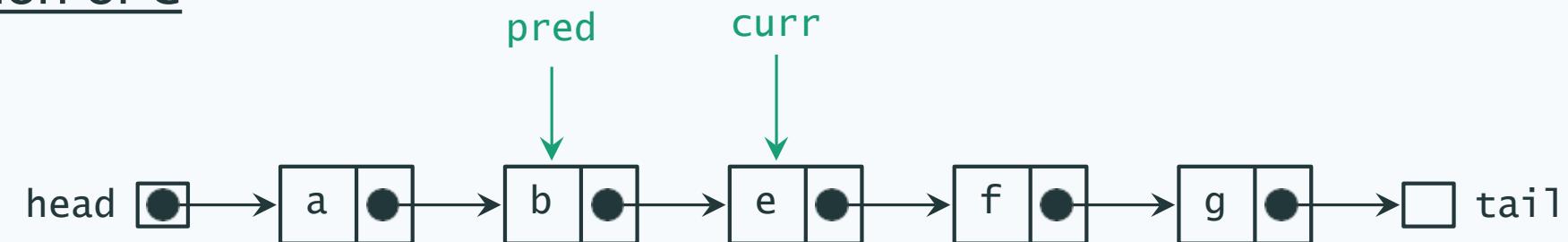
The method `find` implements this frequently used operation of finding the **position** of a key inside a list

The position of key is a **pair** (`pred, curr`) of adjacent nodes, such that

$$\text{pred}.\text{key}() < \text{key} \leq \text{curr}.\text{key}()$$

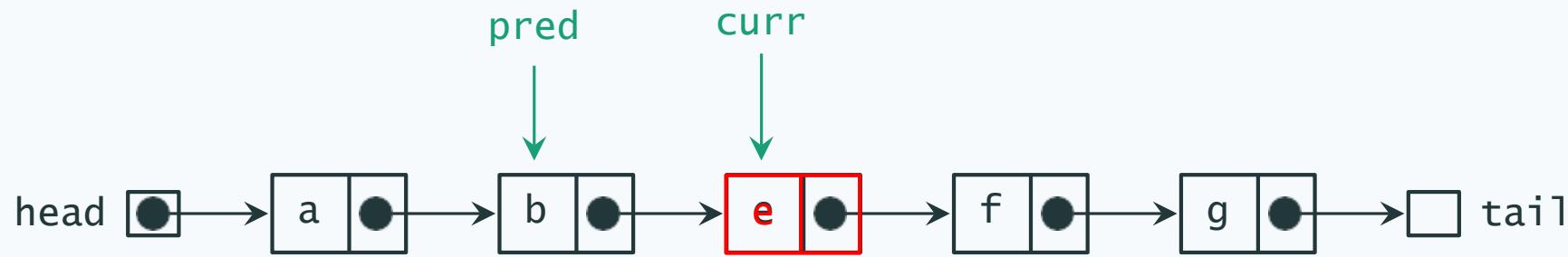
For example, the position of c

in the following list is:



Thanks to the boundary keys chosen for head and tail, searching for **any value key** returns a **valid position** in the list

# Finding a position inside a list

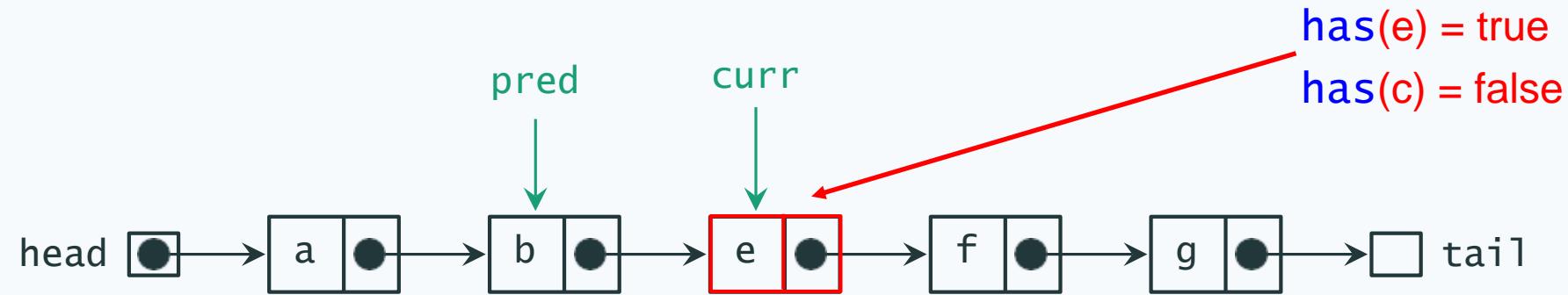


```
// first position from 'start' whose key is no smaller than 'key'  
protected Node<T>, Node<T> find(Node<T> start, int key) {  
    Node<T> pred, curr; // predecessor and current node in iteration  
    curr = start; // from start node  
    do {  
        pred = curr; curr = curr.next(); // move to next node  
    } while (curr.key() < key); // until curr.key >= key  
    return (pred,curr); // return position
```

pseudo-code for: **new** Position<T>(pred,curr)

# Sequential set: method `has`

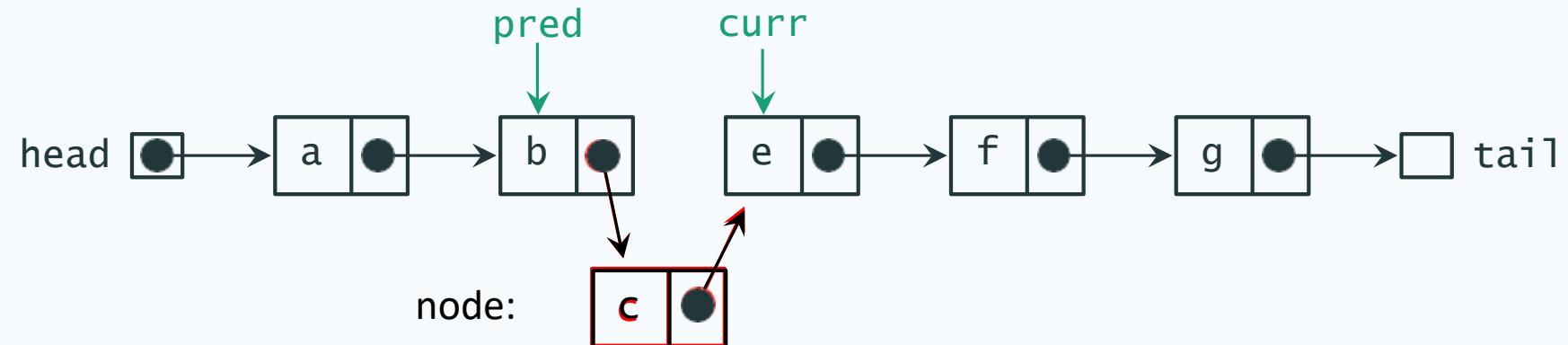
A set `has item` if and only if `item` is (equal to) the first element in the set whose key is greater than or equal to `item`'s



```
// is 'item' in set?  
public boolean has(T item) {  
    int key = item.key(); // item's key  
    // find position of key from head:  
    Node<T> pred, curr = find(head, key);  
    // curr.key() >= key  
    return curr.key() == key; // item can only appear here!  
}
```

# Sequential set: method **add**

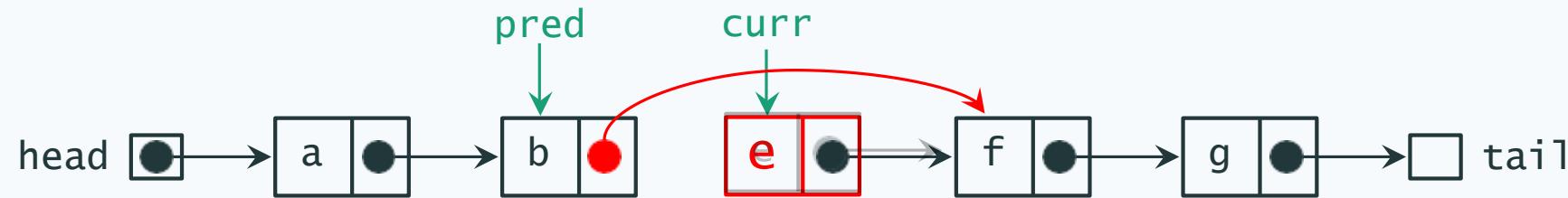
A new item must be **added between** pred and curr, where (pred, curr) is item's **position** in the list



```
public boolean add(T item) {  
    Node<T> node = new Node<>(item);           // new node  
    Node<T> pred, curr = find(head, item.key()); // curr.key >= item.key()  
    if (curr.key() == item.key())                // item already in set  
        return false;  
    else                                         // item not in set: add node between pred and curr  
    {  
        node.setNext(curr);  
        pred.setNext(node);  
        return true;  
    }  
}
```

# Sequential set: method `remove`

An element `item` is **removed** from a set by redirecting `pred.next` to skip over `curr`, where `(pred, curr)` is item's **position** in the list

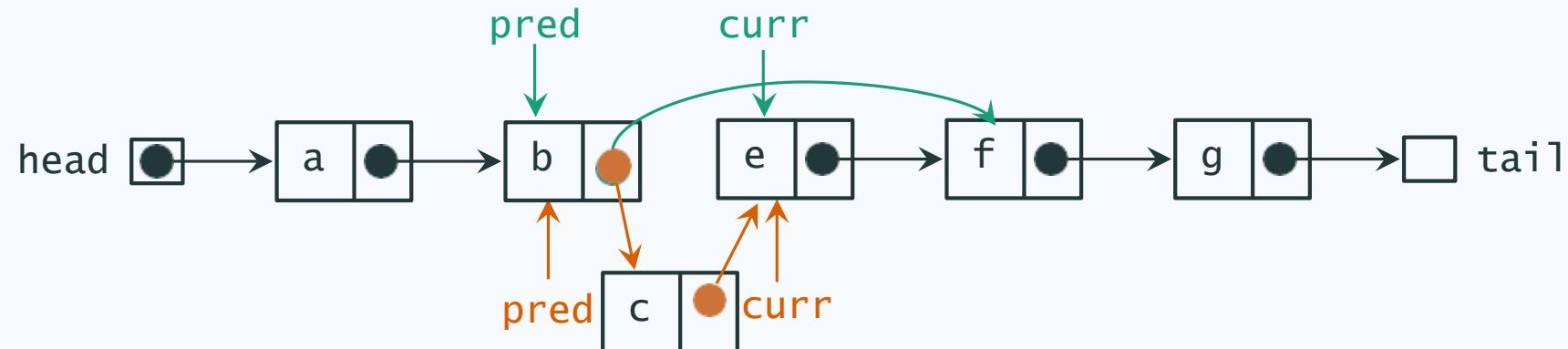


```
public boolean remove(T item) {
    Node<T> pred, curr = find(head, item.key());
    // curr.key() >= item.key()
    if (curr.key() > item.key()) return false; // item not in set
    else // item in set: remove node curr
    {
        pred.setNext(curr.next());
        return true;
    }
}
```

# Sequential set does not work under concurrency

If **multiple threads** are active on the same instance of `sequentialSet`, they can easily **interfere** with each other's operations (and possibly leave the set in an inconsistent state)

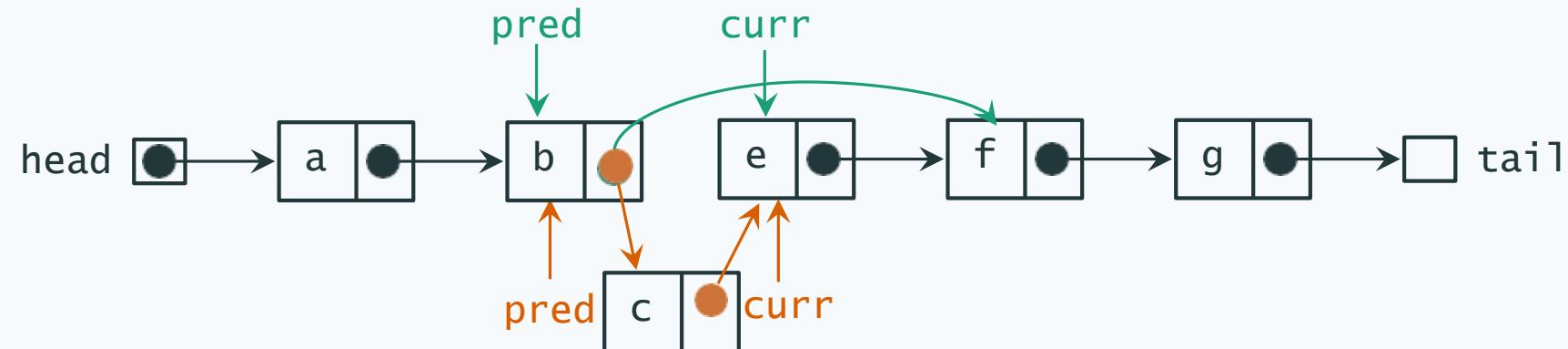
For example, if **thread *t*** runs `remove(e)` while **thread *u*** runs `add(c)`: in some interleavings, **remove is reverted**:



# Sequential set does not work under concurrency

If **multiple threads** are active on the same instance of `sequentialSet`, they can easily **interfere** with each other's operations (and possibly leave the set in an inconsistent state)

For example, if **thread *t*** runs `remove(e)` while **thread *u*** runs `add(c)`: in some interleavings, **remove is reverted**:



# Parallel linked sets

# Parallel linked sets

## Coarse grained locking

# Concurrent set with coarse-grained locking

A straightforward way to make `SequentialSet` work correctly under concurrency is using a `lock` to ensure that **at most one thread at a time** is operating on the structure

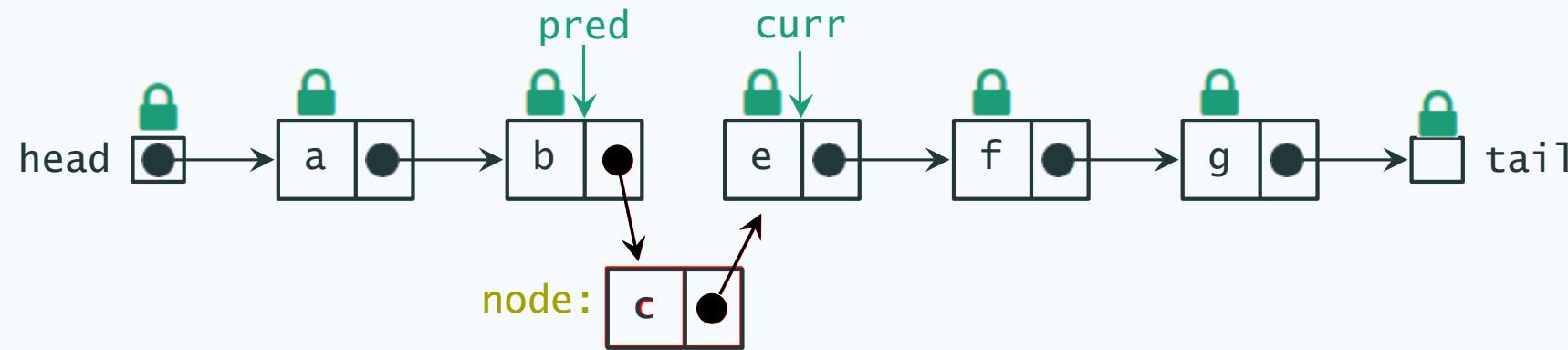
```
class CoarseSet<T> extends SequentialSet<T>
{
    // Lock controlling access to the whole set
    private Lock lock = new ReentrantLock();

    // overriding of add, remove, and has
```

Every method `add`, `remove`, and `has` simply works as follows:

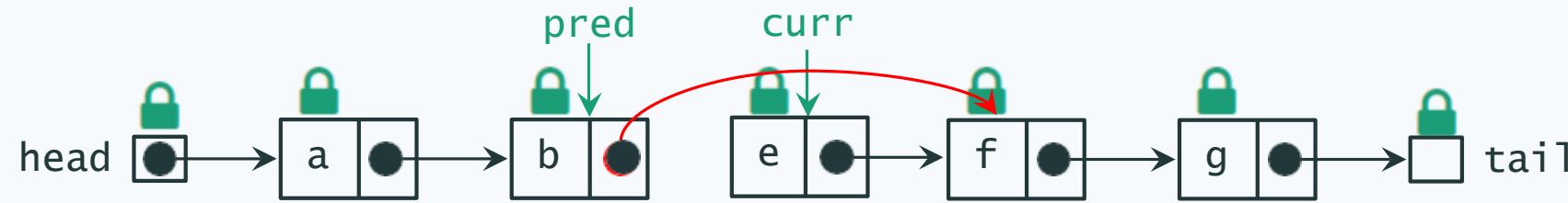
1. acquires the `lock` on the set
2. performs the `operation` as in `SequentialSet`
3. releases the `lock` on the set

# Coarse-locking set: method **add**



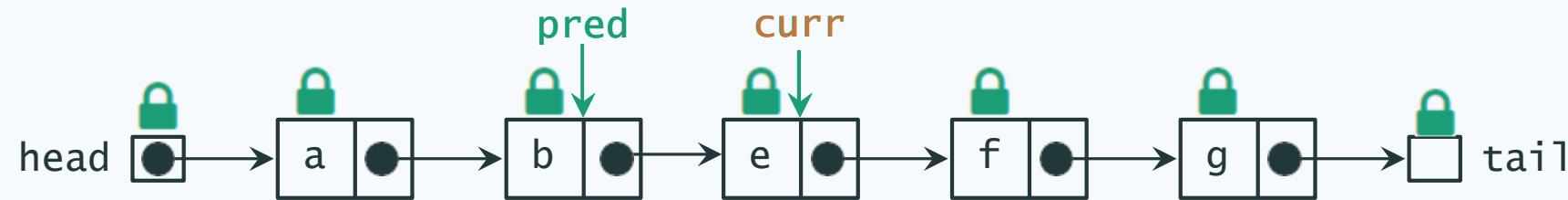
```
public boolean add(T item) {  
    lock.lock(); // lock whole set  
    try {  
        return super.add(item); // execute 'add' while locking  
    } finally {  
        lock.unlock(); // done: release lock  
    }  
}
```

# Coarse-locking set: method **remove**



```
public boolean remove(T item) {  
    lock.lock(); // Lock whole set  
    try {  
        return super.remove(item); // execute 'remove' while locking  
    } finally {  
        lock.unlock(); // done: release lock  
    }  
}
```

# Coarse-locking set: method **has**



```
public boolean has(T item) {  
    lock.lock(); // lock whole set  
    try {  
        return super.has(item); // execute 'has' while locking  
    } finally {  
        lock.unlock(); // done: release lock  
    }  
}
```

# Coarse-locking set: pros and cons

## Pros:

- obviously correct – it avoids race conditions and deadlocks
- if the lock is fair, so is access to the set
- if contention is low (not many threads accessing the set concurrently), coarseSet is quite efficient

## Cons:

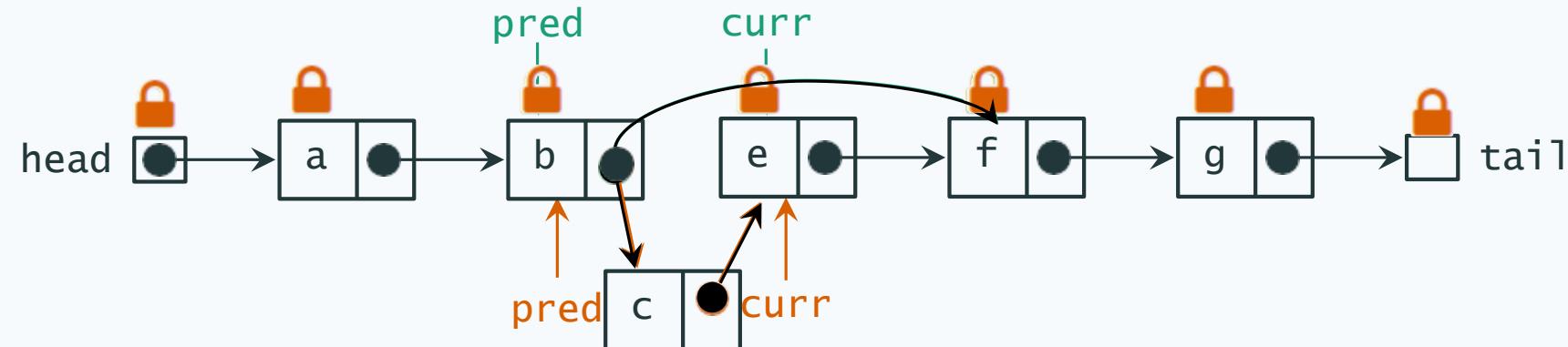
- access to the set is essentially sequential – missing opportunities for parallelization
- if contention is high (many threads accessing the set concurrently), coarseSet is quite slow

# Locking after finding?

Can we reduce the size of the critical sections by executing `find` without locking, and then acquiring the lock only before modifying the list?

**No**, because the list may be modified between when a thread performs `find` and when it acquires the lock

For example, suppose **thread *t*** runs `remove(e)` while **thread *u*** runs `add(c)`, and ***t*** acquires the lock first:



# Parallel linked sets

## Fine grained locking

# Concurrent set with fine-grained locking

Rather than locking the whole linked list at once, we add a **lock** to each node

Then, threads only **lock the individual nodes** on which they are operating

```
public class FineSet<T> extends SequentialSet<T>
{
    // empty set
    public FineSet() {
        head = new LockableNode<>(Integer.MIN_VALUE);    // smallest key
        tail = new LockableNode<>(Integer.MAX_VALUE);    // largest key
        head.setNext(tail);
    }
    // overriding of find, add, remove, and has
```

# Nodes in a fine-locking set

Each node includes a **lock object**, and `lock` and `unlock` methods that access the lock

```
class LockableNode<T> extends SequentialNode<T>
{
    private Lock lock = new ReentrantLock();

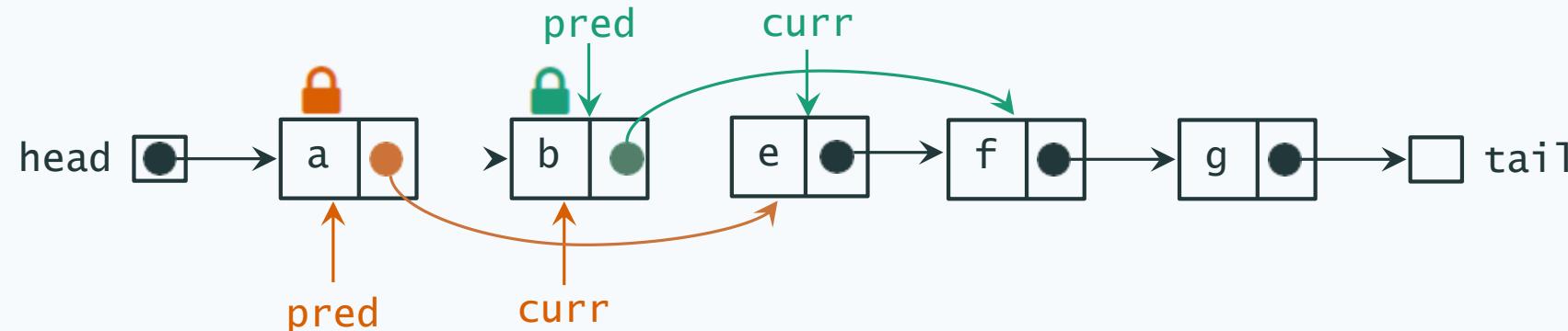
    void lock() { lock.lock(); } // lock node
    void unlock() { lock.unlock(); } // unlock node
}
```

# How many nodes do we have to lock?

We have seen (in coarseSet) that we have to lock as soon as we start executing find. Thus, we start locking the head node and pass the lock along the chain of nodes

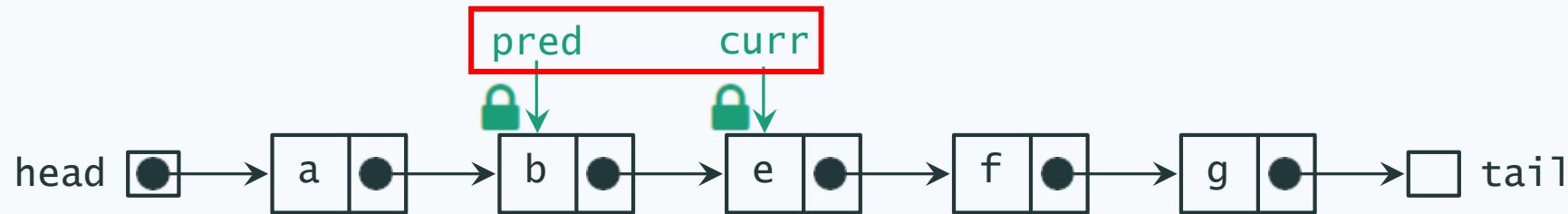
How many nodes do we have to hold locked at once? Even though pred's node is the only node that is actually modified, only locking pred is not enough

For example, if thread *t* runs remove(e) while thread *u* runs remove(b), it may happen that only b's removal takes place:



Problem: we may lock both pred and curr (pred) at once

# Fine-locking set: method **find** (First Attempt!)



```

// find while locking pred and curr, return locked position
protected Node<T>, Node<T> find(Node<T> start, int key) {
    Node<T> pred, curr;                                // predecessor and current node in iteration
    pred = start;                                       // from start node
    curr = start.next();
    pred.lock();                                     // lock pred node
    curr.lock();                                     // lock curr node
    while (curr.key < key) {
        pred.unlock();                               // unlock pred node
        pred = curr;
        curr = curr.next();                          // move to next node
        curr.lock();                                 // lock next node
    }                                                 // until curr.key >= key
    return (pred, curr); /\ return position
}

```

pseudo-code for: **new Position<T>(pred, curr)**

Does it work in all cases?

NO!

A thread may interleave here and remove the current node before the lock is performed

# Fine-locking set: method **find**

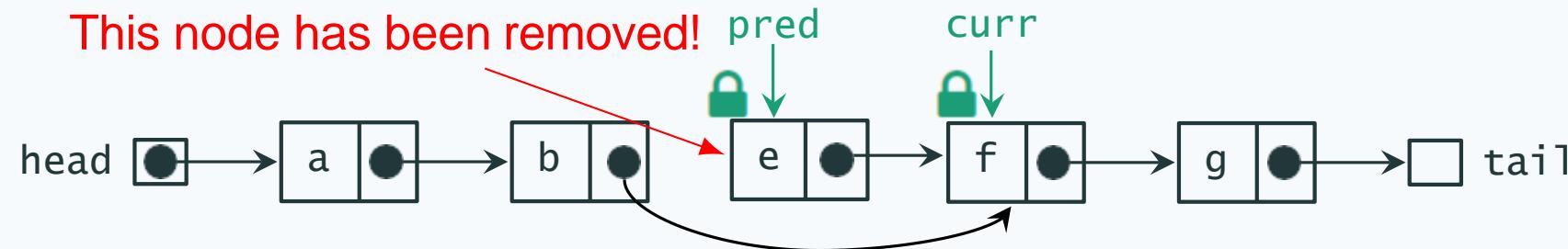
Now the removal cannot take place since the only way to remove the current node is by having a lock on both pred and curr (but the current node holds the lock on pred so no other node can have it)

```
// find while locking pred and curr, return locked position
protected Node<T>, Node<T> find(Node<T> start, int key) {
    Node<T> pred, curr;                                // predecessor and current node in iteration
    pred = start;                                       // from start node
    pred.lock();                                         // lock pred node
    curr = start.next();
    curr.lock();                                         // lock curr node
    while (curr.key < key) {
        pred.unlock();                                 // unlock pred node
        pred = curr;
        curr = curr.next();
        curr.lock();                                   // lock next node
    }                                                       // until curr.key >= key
    return (pred, curr); // return position
}
```

# Hand-over-hand locking

The lock acquisition protocol used by `find` in `FineSet` is called **hand-over-hand locking** or **lock coupling**

- Always keep at least one node locked to prevent interference between threads; **otherwise:**

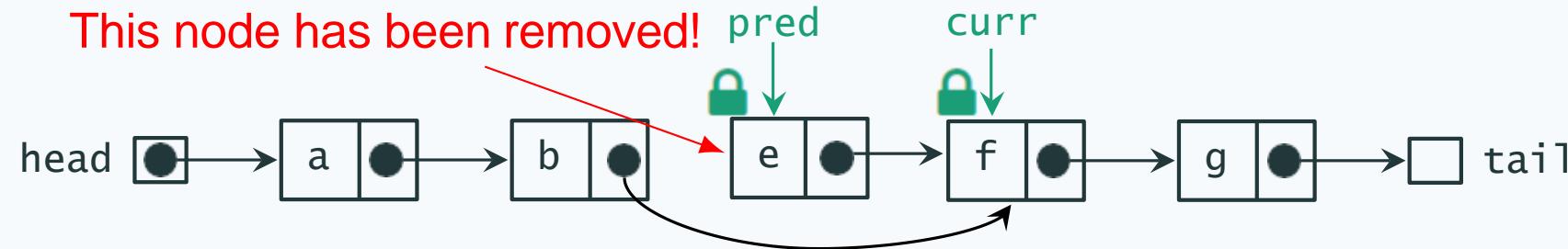


- Locking **two nodes at once** is sufficient to **prevent** problems with **conflicting operations**: threads proceed along the linked list in order, without one thread “overtaking” another thread that is further out
- The protocol ensures locks are acquired by all threads in the same order, **avoiding deadlocks**

# Hand-over-hand locking

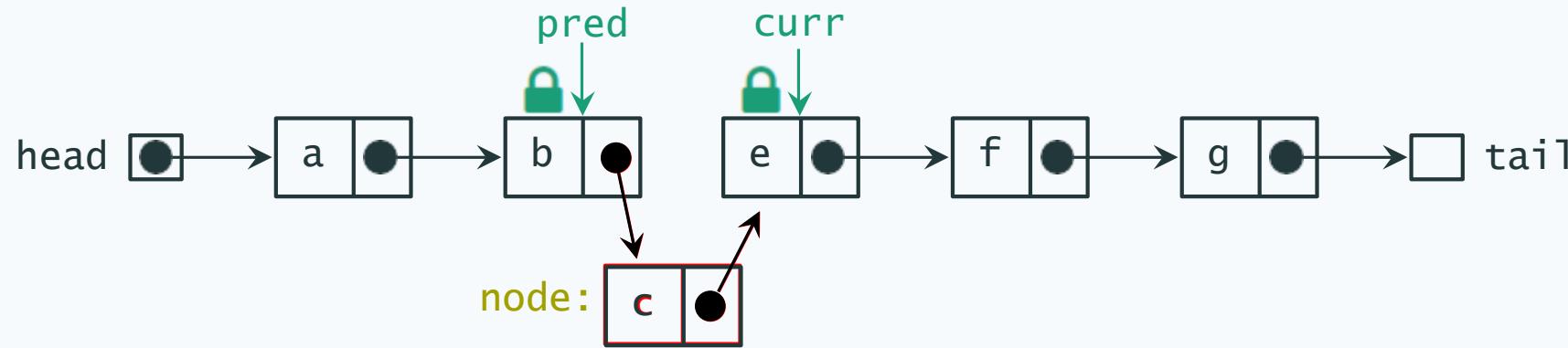
The lock acquisition protocol used by `find` in `FineSet` is called **hand-over-hand locking** or **lock coupling**

- Always keep at least one node locked to prevent interference between threads; **otherwise:**



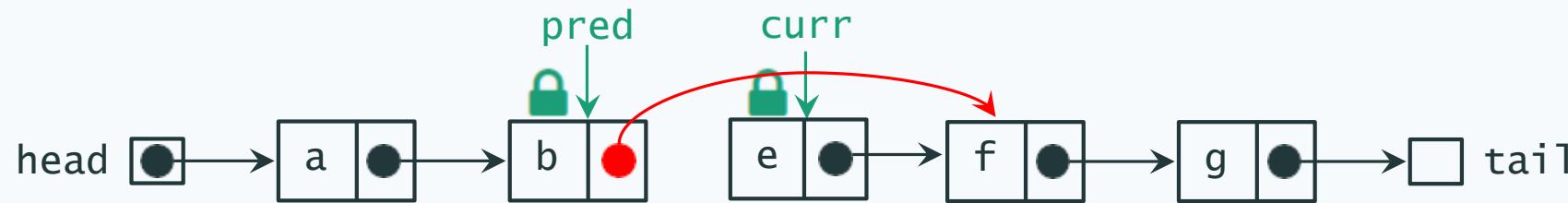
- Locking **two nodes at once** is sufficient to **prevent** problems with **conflicting operations**: threads proceed along the linked list in order, without one thread “overtaking” another thread that is further out
- The protocol ensures locks are acquired by all threads in the same order, **avoiding deadlocks**

# Fine-locking set: method **add**



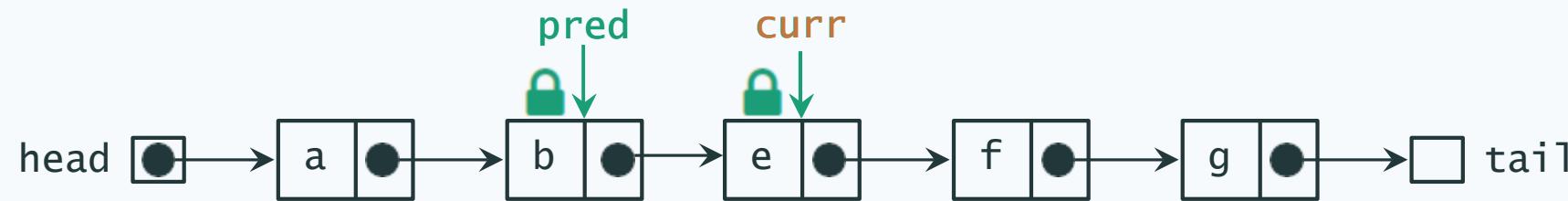
```
public boolean add(T item) {  
    Node<T> node = new LockableNode<>(item); // new node  
    try { // find with hand-over-hand locking  
        // the first position such that curr.key() >= item.key()  
        Node<T> pred, curr = find(head, item.key()); // locking  
        ... // add node as in SequentialSet, while locking  
    } finally { pred.unlock(); curr.unlock(); } // done: unlocking  
}
```

# Fine-locking set: method `remove`



```
public boolean remove(T item) {  
    try { // find with hand-over-hand locking  
        // the first position such that curr.key() >= item.key()  
        Node<T> pred, curr = find(head, item.key()); // locking  
        ... // remove node as in SequentialSet, while locking  
    } finally { pred.unlock(); curr.unlock(); } // done: unlocking  
}
```

# Fine-locking set: method **has**



```
public boolean has(T item) {  
    try { // find with hand-over-hand locking  
        // the first position such that curr.key() >= item.key()  
        Node<T> pred, curr = find(head, item.key()); // locking  
        ... // check node as in SequentialSet, while locking  
    } finally { pred.unlock(); curr.unlock(); } // done: unlocking  
}
```

# Fine-locking set: pros and cons

## Pros:

- if locks are fair, so is access to the set, because threads proceed along the list one after the other without changing order
- threads operating on disjoint portions of the list may be able to operate in parallel

## Cons:

- it is still possible that one thread prevents another thread from operating in parallel on a disjoint portion of the list – for example, if one thread wants to access the end of the list but another thread blocks it while locking the beginning of the list
- the hand-over-hand locking protocol may be quite slow, as it involves a significant number of lock operations

# Parallel linked sets

## Optimistic locking

# Concurrent set with optimistic locking

Let us revisit the idea of performing `find without locking`

We have seen that problems may occur if the list is modified between when a threads finds a position and when it acquires locks on that position

Thus, we validate a position `after finding it` and while the nodes are locked, to verify that no interference took place

```
public class OptimisticSet<T> extends SequentialSet<T>
{
    public FindSet()
    { head = new ReadwriteNode<>(Integer.MIN_VALUE);           // smallest key
      tail = new ReadwriteNode<>(Integer.MAX_VALUE);           // largest key
      head.setNext(tail); }

    // is (pred, curr) a valid position?
    protected boolean valid(Node<T> pred, Node<T> curr) // ...
}

// overriding of find, add, remove, and has
```

# Nodes in an optimistic-locking set

Since we need to be able to **follow** the chain of `next` references **without locking**, attribute `next` must be declared **volatile** in Java – so that modifications to it (which occur while the node is locked) are propagated to all threads (even if they have not locked a node)

- Other than for this detail, a `ReadWriteNode` is the same as a `LockableNode`
- With a little abuse of notation, we can pretend that `ReadWriteNode` inherits from `LockableNode` and overrides its `next` attribute

Overriding of attributes is however not possible in Java (shadowing takes place instead); the actual implementation that we make available does not reuse `LockableNode`'s code through inheritance

```
class ReadWriteNode<T> extends LockableNode<T>
{
    private volatile Node<T> next;           // next node in chain
}
```

# Delayed locking as optimistic locking

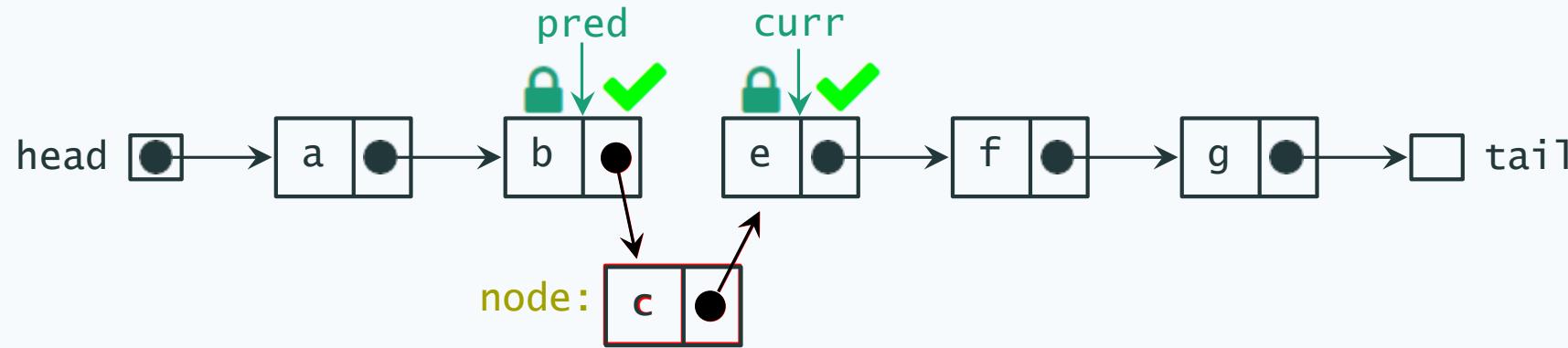
In optimisticset, operations work as follows:

1. **find** the item's position inside the list without locking – as in SequentialSet
2. **lock** the position's nodes pred and curr
3. **validate** the position while the nodes are locked:
  - 3.1 if the position is valid, **perform the operation** while the nodes are locked, then release locks
  - 3.2 if the position is invalid, release locks and **repeat the operation** from scratch

This approach is **optimistic** because it works well when validation is often successful (so we don't have to repeat operations)

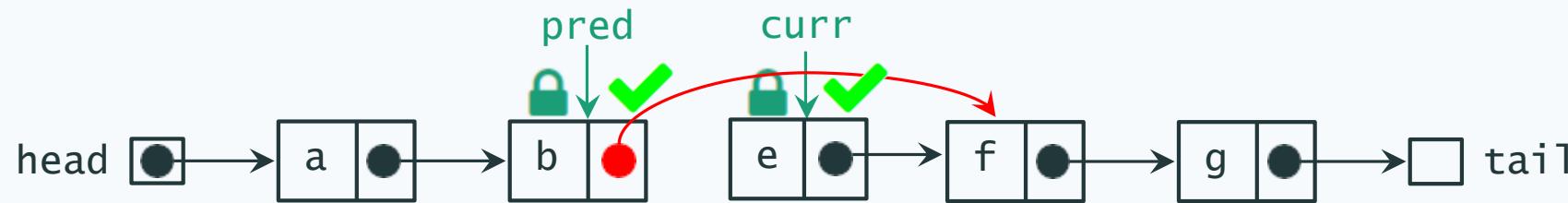


# Optimistic set: method add



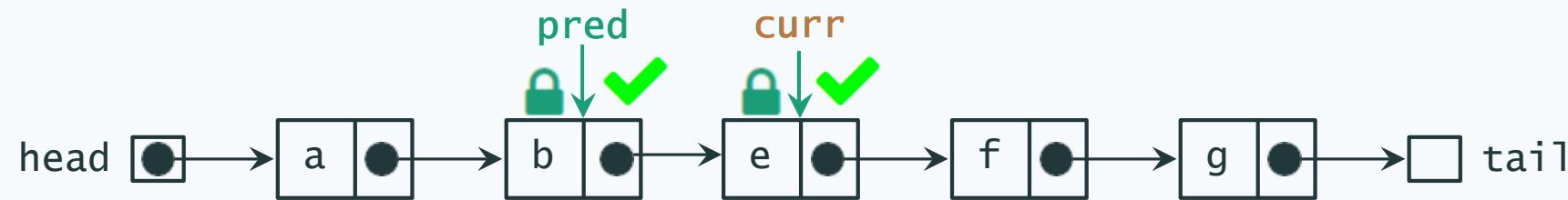
```
public boolean add(T item) {  
    Node<T> node = new ReadwriteNode<>(item); // new node  
    do { Node<T> pred, curr = find(head, item.key()); // no locking  
        pred.lock(); curr.lock(); // now lock position  
        try { // if position still valid, while locked:  
            if (valid(pred, curr)) { ... } // physically add node  
        } finally { pred.unlock(); curr.unlock(); } // done: unlock  
    } while (true); // if not valid: try again!  
}
```

# Optimistic set: method `remove`



```
public boolean remove(T item) {  
    do { Node<T> pred, curr = find(head, item.key());           // no locking  
        pred.lock(); curr.lock();                                // now lock position  
        try { // if position still valid, while locked:  
            if (valid(pred, curr)) { ... }                      // physically remove node  
        } finally { pred.unlock(); curr.unlock(); }               // done: unlock  
    } while (true);                                            // if not valid: try again!  
}
```

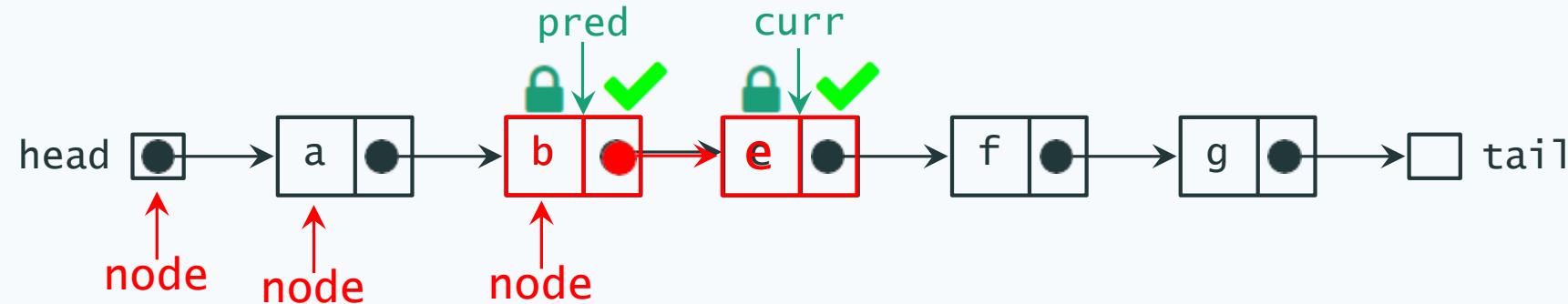
# Optimistic set: method **has**



```
public boolean has(T item) {
    do { Node<T> pred, curr = find(head, item.key());                                // no locking
          pred.lock(); curr.lock();                                                 // now lock position
          try { // if position still valid, check key while locked
              if (valid(pred, curr)) return curr.key() == item.key();
          } finally { pred.unlock(); curr.unlock(); }                                // done: unlock
    } while (true);                                                               // if not valid: try again!
}
```

# Optimistic set: validating a position

Validation goes through the nodes until it reaches the given position



// Is pred reachable from head, and does it point to curr?

```
protected boolean valid(Node<T> pred, Node<T> curr) {  
    Node<T> node = head;                                // start from head  
    while (node.key() <= pred.key()) {                    // does pred point to curr?  
        if (node == pred) return pred.next() == curr;  
        node = node.next();                               // continue to the next node  
    } // until node.pred > pred.key  
    return false;                                         // pred could not be reached  
} // or does not point to curr
```

# How validation works

What can happen **between** the time when a thread **finds** a position (pred, curr) **and** when it **locks** nodes pred and curr?

- Node pred is removed: **validation fails** because pred is not reachable
- Node curr is removed: **validation fails** because pred does not point to curr
- A node is added between pred and curr: **validation fails** because pred does not point to curr
- Any other modification of the set: **validation succeeds** because operations leave the set in a consistent state

# Is validation safe?

What happens if the set is being **modified while** a thread is **validating** a locked position (pred, curr)?

- If a node following curr is modified: validation is not affected because it only goes up until curr
- If a node n before pred is removed: validation succeeds even if it goes through n, since n still leads back to pred
- If a node n is added before pred: validation succeeds even if it skips over n

# Optimistic-locking set: pros and cons

## Pros:

- threads operating on disjoint portions of the list can operate in parallel
- when validation often succeeds, there is much less locking involved than in Fineset

## Cons:

- `optimisticSet` is not starvation free: a thread  $t$  may fail validation forever if other threads keep removing and adding pred/curr between when  $t$  performs `find` and when it locks pred and curr
- if traversing the list twice without locking is not significantly faster than traversing it once with locking, `optimisticSet` does not have a clear advantage over `Fineset`

# Parallel linked sets

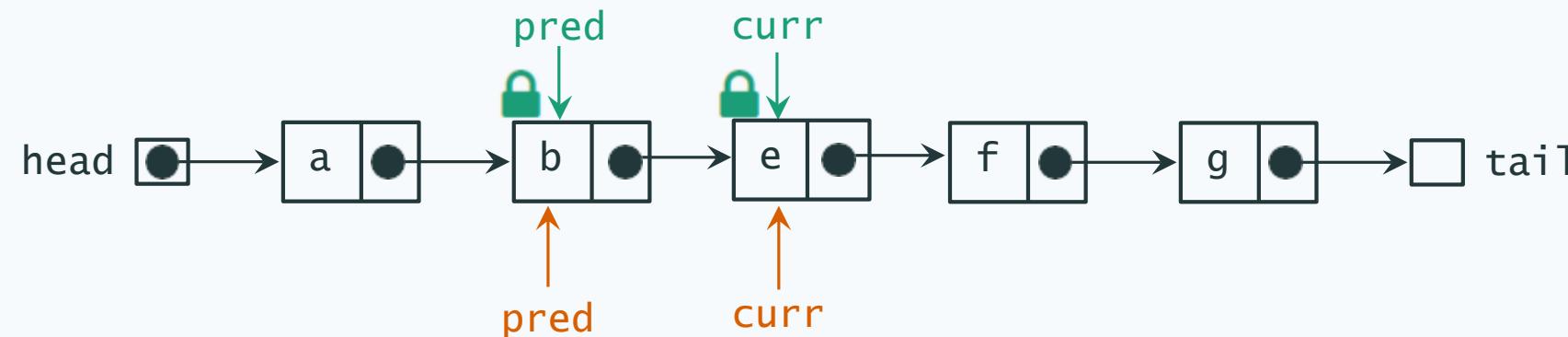
## Lazy node removal

# Testing membership without locking

In many applications, `has` is executed **many more times** than `add` and `remove`.  
Can `has` work correctly **without locking**?

Problems may occur if another thread removes `curr` between `find` and `has`'s check:  
since `remove` is not atomic without locking, if `has` does not acquire locks it may not  
notice that `curr` is being removed

For example, if **thread *t*** runs `remove(e)` while **thread *u*** runs `has(e)` without locking, ***u***  
may incorrectly think that `e` is in the list even if ***t*** is about to remove it – that is **thread *t***  
is in its critical section:



# Nodes in a lazy-removal set

We need a way to **atomically** share the **information** that a node is being **removed**, but without locking

To this end, each node includes a **flag valid** with setters and getters:

- `valid() == true`: the node is part of the set
- `valid() == false`: the node is being (or has been) removed

```
class ValidatedNode<T> extends ReadwriteNode<T>
{
    private volatile boolean valid;

    boolean valid() { return valid; }          // is node valid?
    void setValid() { valid = true; }          // mark valid
    void setInvalid() { valid = false; }        // mark invalid
}
```

Nodes of type `ValidatedNode` can also be locked, since `ValidatedNode` inherits from `ReadwriteNode`

# Concurrent set with lazy node removal

In a **lazy set**:

- Validation only needs to check the mark valid
- Operation `remove` marks a node invalid before removing it
- Operation `has` is lock-free
- Operation `add` works as in `optimisticSet`

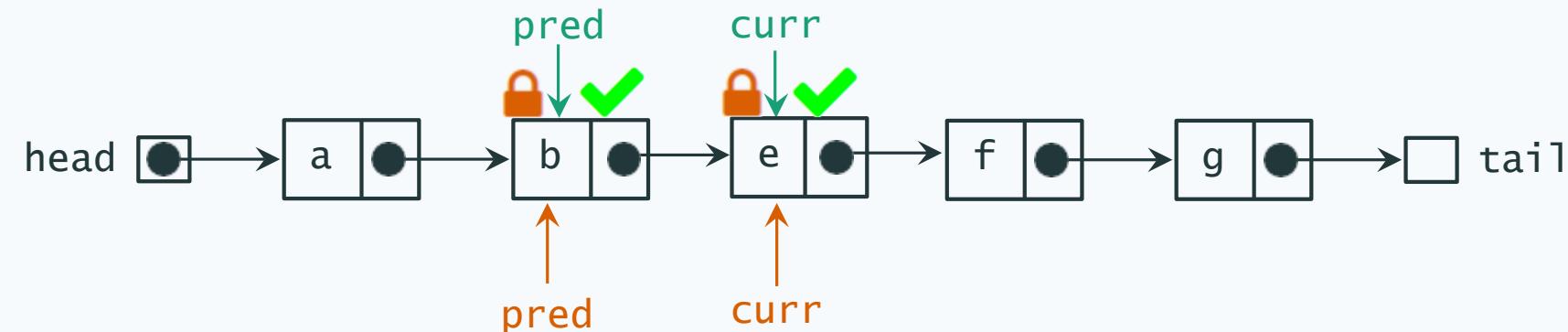
```
public class LazySet<T> extends OptimisticSet<T>
{
    public LazySet() {
        head = new validatedNode<>(Integer.MIN_VALUE); // smallest key
        tail = new validatedNode<>(Integer.MAX_VALUE); // largest key
        head.setNext(tail);
    }
    // overriding of valid, remove, and has
```

# Lazy set: validating a position

Validation becomes a constant-time operation:

- Node pred is **reachable** from the head iff it has not been removed iff it is marked **valid**
- Node curr follows pred in the list iff pred.next() == curr **and** curr is marked **valid**

Scenario: *t*'s validation of curr **succeeds**:



*// is pred reachable from head, and does it point to curr?*

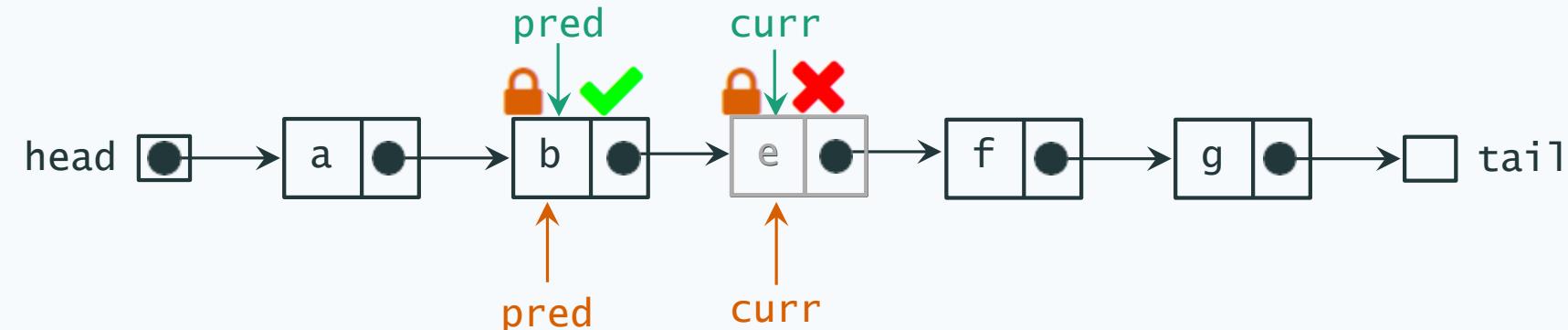
```
protected boolean valid(Node<T> pred, Node<T> curr) {  
    return pred.valid() && curr.valid() && pred.next() == curr;  
}
```

# Lazy set: validating a position

Validation becomes a constant-time operation:

- Node `pred` is **reachable** from the head iff it has not been removed iff it is marked **valid**
- Node `curr` follows `pred` in the list iff `pred.next() == curr` **and** `curr` is marked **valid**

Scenario: `t`'s validation of `curr` **fails**:



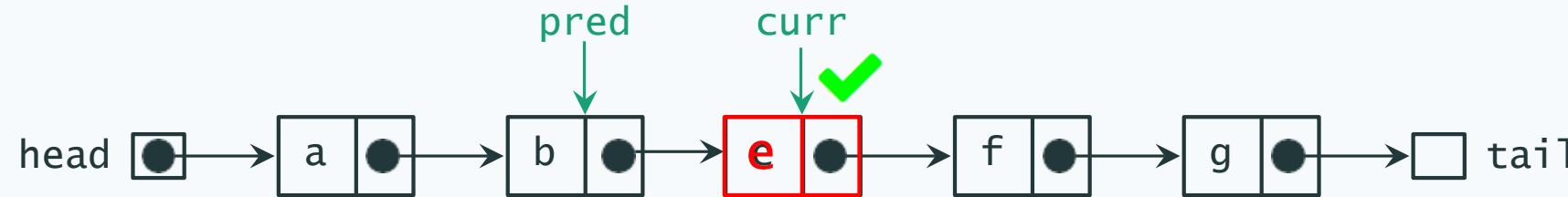
*// is pred reachable from head, and does it point to curr?*

```

protected boolean valid(Node<T> pred, Node<T> curr) {
    return pred.valid() && curr.valid() && pred.next() == curr;
}
  
```

# Lazy set: method **has**

Method **has** runs **without locking**: it finds the position (**pred**, **curr**), validates **curr**, and checks whether **curr**'s key is equal to **item**'s

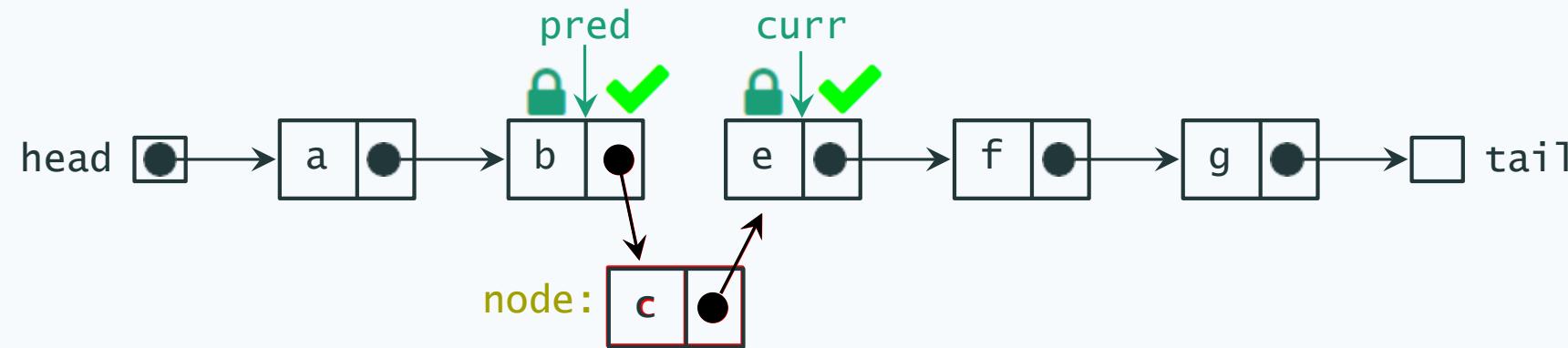


```
public boolean has(T item) {  
    // find position without locking  
    Node<T> pred, curr = find(head, item.key());  
    // check validity and item without locking  
    return curr.valid() && curr.key() == item.key();  
}
```

Method **find** may traverse **invalid nodes**; this does not prevent it from eventually reaching **all valid** nodes in the list

# Lazy set: method **add**

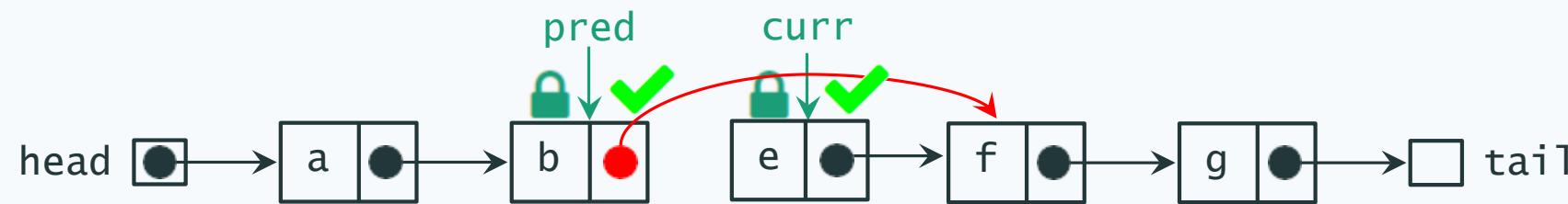
Method add works **as in optimisticset**, but using the overridden version of valid – which works in constant time



# Lazy set: method `remove`

After finding the position of a node to be removed, the **actual removal** consists of two steps

1. **logical removal**: mark the node to be removed as invalid
2. **physical removal**: skip over the node by redirecting its predecessor's next



This removal is **lazy** because logical and physical removal may be done at different times: after a node has been logically removed, every thread is aware that it should not be considered part of the list

# Lazy set: method `remove`

```
public boolean remove(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // no locking  
        pred.lock(); curr.lock(); // now lock position  
        try { // if position still valid, while locking:  
            if (valid(pred, curr)) {  
                if (curr.key() != item.key())  
                    return false; // item not in the set  
                else { // item in the set at curr: remove it  
                    curr.setInvalid(); // logical removal  
                    pred.setNext(curr.next()); // physical removal  
                    return true;  
                }  
            }  
        } finally { pred.unlock(); curr.unlock(); } // done: unlock  
    } while (true); // if not valid: try again!  
}
```

# Lazy-removal set: pros and cons

## Pros:

- validation is constant time
- membership checking does not require any locking – it's even **wait-free** (it traverses the list once without locking)
- physical removal of logically removed nodes could be **batched** and performed when convenient – thus reducing the number of times the physical chain of nodes is changed, in turn reducing the expensive propagation of information between threads

## Cons:

- operations `add` and `remove` still require locking (as in `optimisticset`), which may reduce the amount of parallelism

# Parallel linked sets

Lock free access

# Atomic references

To implement a set that is correct under concurrent access **without using any locks** we need to rely on **synchronization primitives** more **powerful** than just reading and writing shared variables

We are going to use a variant of the **compare-and-set** operation

```
class AtomicReference<V> {  
    V get();                      // current reference  
    void set(V newRef);           // set reference to newRef  
  
    // if reference == expectRef, set to newRef and return true  
    // otherwise, do not change reference and return false  
    boolean compareAndSet(V expectRef, V newRef);  
}
```

# Atomic lock-free access: first naive attempt

As a [first attempt](#), we make attribute next of type `AtomicReference<Node<T>>` and use `compareAndSet` to update it: if one thread changes next when another thread is also trying to change it, we [repeat](#) the operation

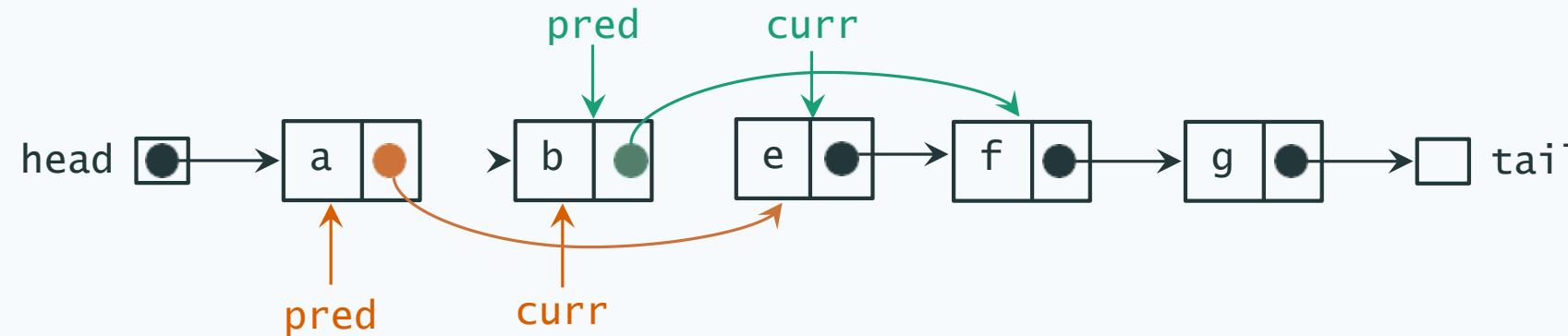
An implementation of `remove()` following this idea:

```
public boolean remove(T item) {
    boolean done;
    do {
        Node<T> pred, curr = find(head, item.key());
        if (curr.key() >= item.key()) return false; // item not in set
        else
            // try to remove curr by setting pred.next using compareAndSet
            done = pred.next().compareAndSet(pred.next(), curr.next());
    } while (!done); return true;
}
```

↑  
pred.next may have changed  
when `compareAndSet()` executes

# Atomic lock-free access: first naive attempt

Unfortunately, the first attempt **does not work**: for example, if **thread *t*** runs `remove(e)` while **thread *u*** runs `remove(b)`, it may happen that only b's removal takes place



We have seen a similar problem before: modifications of the list need to have **control** of **both pred and curr** – even if it is only the former node that is actually modified

# Atomic markable references

As in LazySet, nodes can be marked valid or invalid; an invalid node is logically removed. In addition, we need to access the information of both attributes valid and next atomically: every node includes an attribute nextvalid of type AtomicMarkableReference<Node<T>>, which provides methods to both update a reference and mark it, atomically.

```
class AtomicMarkableReference<V> {  
    V, boolean get();           // current reference and mark  
    // if reference == expectRef set mark to newMark and return true  
    // otherwise do not change anything and return false  
    boolean attemptMark(V expectRef, boolean newMark);  
    // if reference == expectRef and mark == expectMark,  
    // set reference to newRef, mark to newMark and return true;  
    // otherwise, do not change anything and return false  
    boolean compareAndSet(V expectRef, V newRef, boolean expectMark, boolean newMark)  
}
```

# Nodes in a lock-free set

Every node has an attribute `nextvalid` typed `AtomicMarkableReference<Node<T>>`

The node interface provides methods to retrieve and conditionally update the current value of `nextvalid`, which includes a reference (corr. to `next`) and a mark (corr. to `valid`)

```

class LockFreeNode<T> extends SequentialNode<T> {

    // reference to next node and validity mark of current node
private AtomicMarkableReference<Node<T>> nextvalid;                                nextvalid
                                                                 =  

    // return next and valid as a pair                                                 (next_node, valid_node)
Node<T>, boolean nextvalid() { return nextvalid.get(); }

Node<T> next()
    { Node<T> next, boolean valid = nextvalid(); return next; }

boolean valid()
    { Node<T> next, boolean valid = nextvalid(); return valid; }
  
```

# Nodes in a lock-free set

Every node has an **attribute nextvalid** typed `AtomicMarkableReference<Node<T>>`

The node interface provides methods to retrieve and conditionally update the current value of `nextvalid`, which includes **a reference** (corr. to `next`) and **a mark** (corr. to `valid`)

```
class LockFreeNode<T> extends SequentialNode<T> {  
  
    // try to set invalid; return true if successful  
    boolean setInvalid()  
    { Node<T> next = next();  
        return nextValid.compareAndSet(next, next, true, false); }  
  
    // try to update to newNext if valid; return true if successful  
    boolean setNextIfValid(Node<T> expectNext, Node<T> newNext)  
    { return nextValid.compareAndSet(expectNext, newNext, true, true); }  
  
    update next only if the node is valid
```

- expectRef
- nextRef
- expectMark
- newMark

# Concurrent set with lock-free access

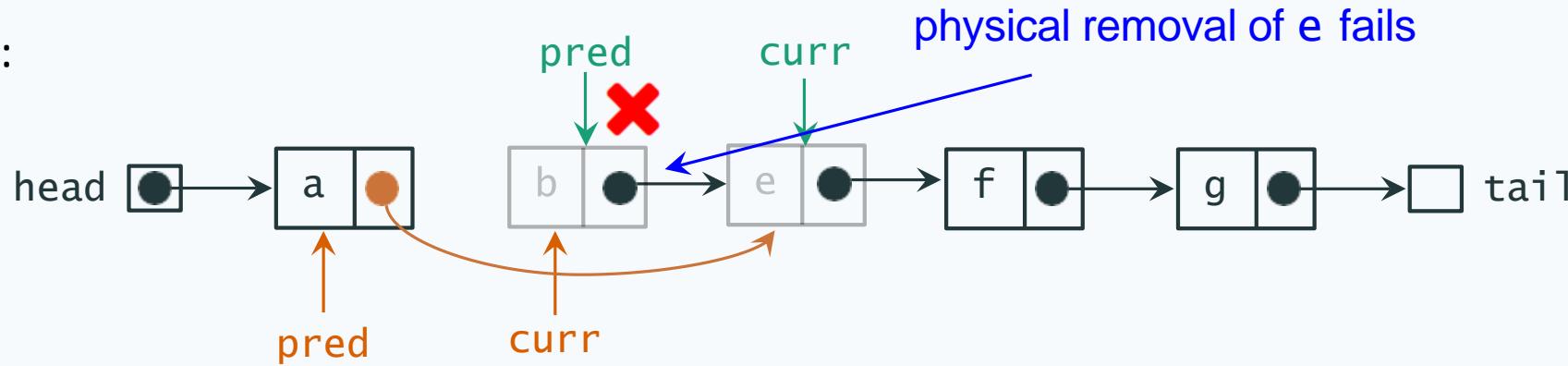
In a lock-free set:

- Operation remove marks a node **invalid** before removing it
- Operations that modify nodes complete successfully **only if** the nodes are **valid** and not **concurrently modified** by another thread
- Failed operations are **repeated until success** (no interference)

```
public class LockFreeSet<T> extends SequentialSet<T>
{
    public LockFreeSet() {
        head = new LockFreeNode<>(Integer.MIN_VALUE); // smallest key
        tail = new LockFreeNode<>(Integer.MAX_VALUE); // largest key
        head.setNext(tail); // unconditionally set next only in new nodes
    }
    // overriding of all methods
```

# Lock-free set: method `remove`

Scenario 1:

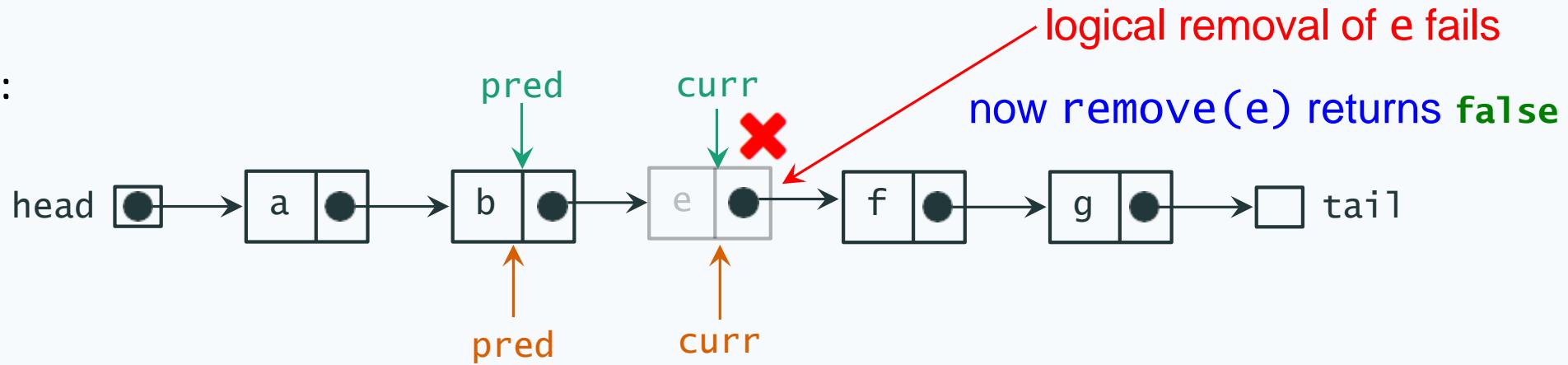


```
public boolean remove(T item) {  
    do { Node<T> pred, curr = find(head, item.key());  
        if (curr.key() != item.key() || !curr.valid()) return false; // not in set or invalid  
        // try to invalidate; try again if node is being modified:  
        if (!curr.setInvalid()) continue;  
        // try once to physically remove curr:  
        pred.setNextIfValid(curr, curr.next());  
        return true;  
    } while (true); // changed during logical removal: try again!  
}
```

physical removal of e  
fails: never mind!

# Lock-free set: method **remove**

Scenario 2:



```
public boolean remove(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // not in set  
        if (curr.key() != item.key() || !curr.valid()) return false;  
        // try to invalidate; try again if node is being modified  
        if (!curr.setInvalid()) continue; // try once to physically remove curr  
        pred.setNextIfValid(curr, curr.next());  
    } while (true);  
    return true;  
} while (true); // changed during logical removal: try again!
```

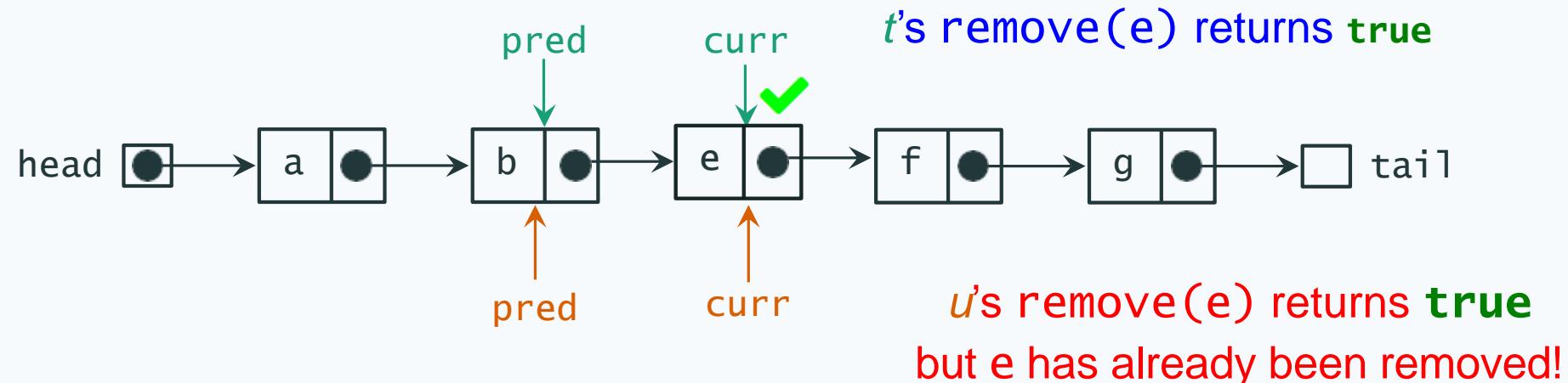
logical removal of e fails: retry!

# Logical removal: only one thread succeeds

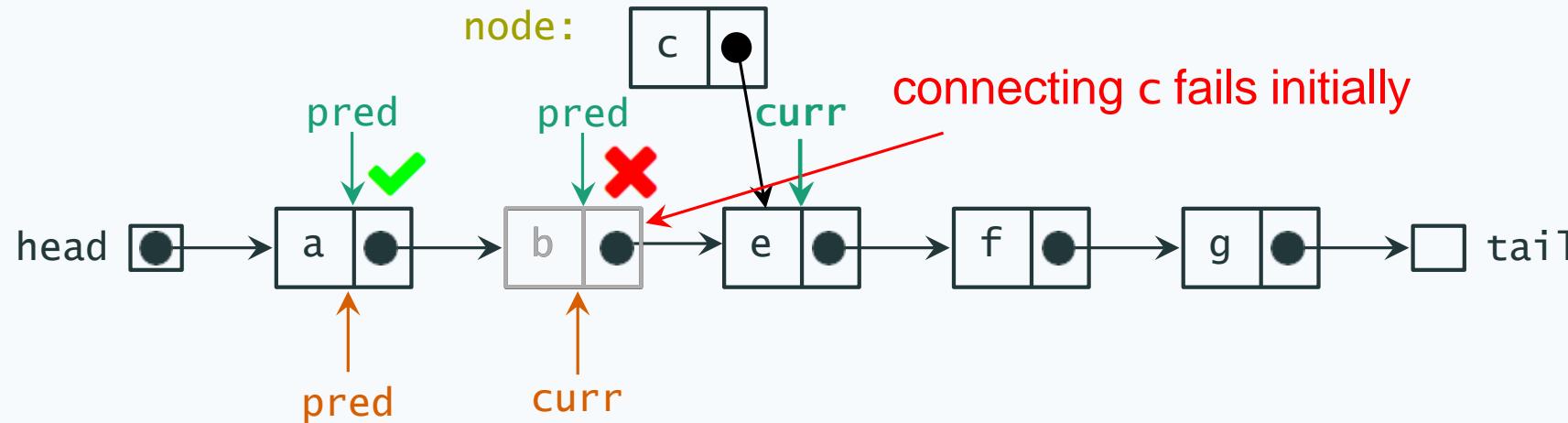
If two threads both try to mark a node invalid, **only one can succeed** – so it is guaranteed that no other thread is touching the node

If this property were not enforced:

- The same element may be removed twice



# Lock-free set: method add

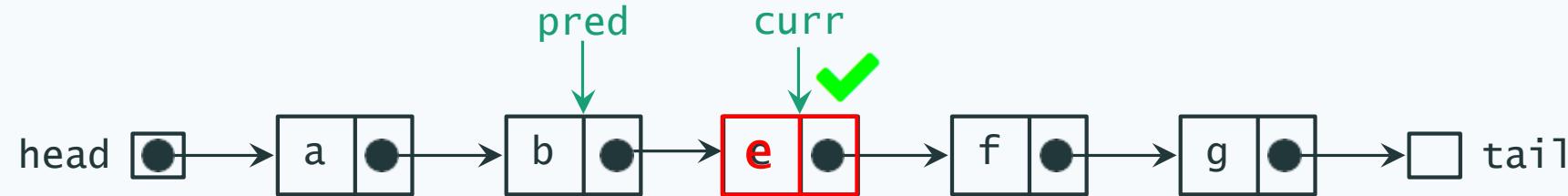


```
public boolean add(T item) {
    do { Node<T> pred, curr = find(head, item.key());
        if (curr.key() == item.key() && curr.valid()) return false; // already in set and valid
        // new node, pointing to curr:
        Node<T> node = new LockFreeNode<T>(item).setNext(curr);
        // if pred valid and points to curr, make it point to node:
        if (pred.setNextIfValid(curr, node)) return true;
    } while (true); // pred changed during add: try again!
}
```

# Lock-free set: method **has**

Method **has** works as in `LazySet`: it finds the position (`pred, curr`), validates `curr`, and checks whether `curr`'s key is equal to `item`'s

Unlike `add` and `remove` (which use a new version of `find`), `has` traverses both valid and invalid nodes, and makes no attempt at removing the latter



```

public boolean has(T item) {
  // find position (use plain search in SequentialSet)
  Node<T> pred, curr = super.find(head, item.key());
  // check validity and item
  return curr.valid() && curr.key() == item.key();
}
  
```

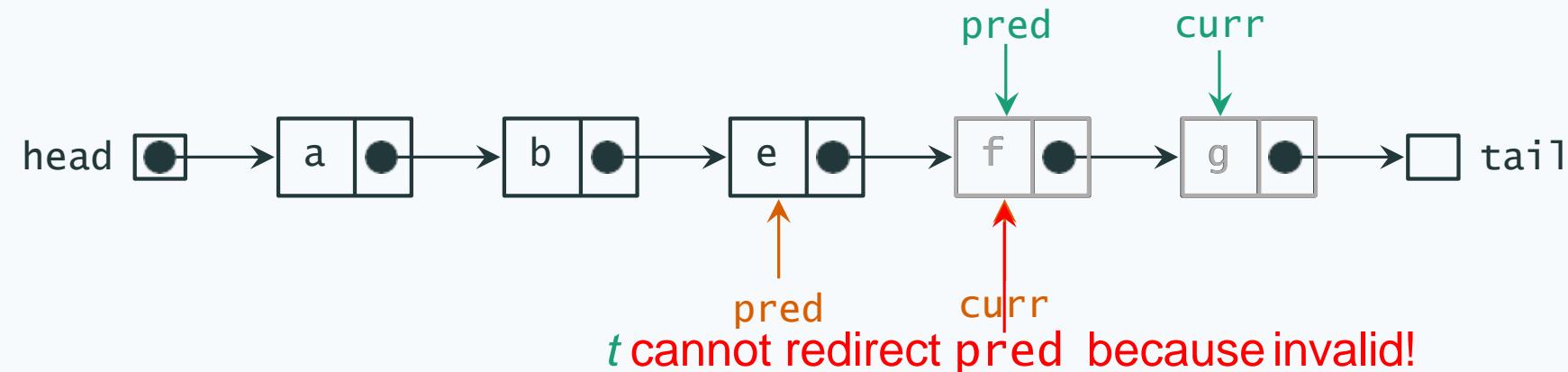
# When to physically remove nodes?

Method `has` does not modify the set, so it can safely traverse valid and invalid nodes without changing the node structure

In contrast, methods `add` and `remove` **physically remove** all logically removed nodes encountered by `find`

This is a convenient time to perform physical removal, because it avoids the buildup of long chains of invalid nodes

For example, the logical removal of nodes `f` and `g` requires `thread t` to physically remove `f` before it can physically remove `g`:



# Lock-free set: how **find** works

Example: A run of `find(k)` that **also physically removes** three invalid nodes



Threads may interfere with `find`, requiring to restart it

In the worst case, **starvation** may occur with a thread continuously restarting `find` while others make progress modifying the list

# Lock-free set: method `find`

We keep track of 3 nodes!

```

protected Node<T>, Node<T> find(Node<T> start, int key) {
  boolean valid;                                // is curr valid?
  Node<T> pred, curr, succ;                    // consecutive nodes in iteration
  retry: do {
    pred = start; curr = start.next();           // from start node
    do { // succ is curr's successor; valid is curr's validity
      succ, valid = curr.nextvalid();
      while (!valid) { // while curr is not valid, try to remove it
        // if pred is modified while trying to redirect it, retry
        if (!pred.setNextIfValid(curr, succ)) continue retry;
        // curr has been physically removed: move to next node
        curr = succ; succ, valid = curr.nextvalid();
      } // now curr is valid (and so is pred)
      if (curr.key() >= key) return (pred, curr);
      pred = curr; curr = succ; // continue search
    } while (true);
  } while (true);
}
  
```

# Lock-free set: pros and cons

## Pros:

- no operations require locking: maximum potential for parallelism
- membership checking does not require any locking – it's even **wait-free** (it traverses the list once without locking)

## Cons:

- the implementation needs **test-and-set-like synchronization primitives**, which have to be supported and come with their own performance costs
- operations add and remove are **lock-free** but **not wait-free**: they may have to repeat operations, and they may be delayed while they physically remove invalid nodes, with the risk of introducing contention on nodes that have been already previously logically deleted

# To lock or not to lock?

Each of the different implementations of concurrent set is the **best choice** for certain **applications** and not for others:

- **CoarseSet** works well with low contention
- **FineSet** works well when threads tend to access the list orderly
- **OptimisticSet** works well to let threads operate on disjoint portions of the list
- **LazySet** works well when batching invalid node removal is convenient
- **LockFreeSet** works well when locking is quite expensive

No many threads accessing the data structure at the same time



© 2016–2019 Carlo A. Furia, Sandro Stucki



Except where otherwise noted, this work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

To view a copy of this license, visit

<http://creativecommons.org/licenses/by-sa/4.0/>.

# Lock-free programming and Software Transactional Memory

TDA384/DIT391

Principles of Concurrent Programming



UNIVERSITY OF  
GOTHENBURG



UNIVERSITY OF  
GOTHENBURG

---

Nir Piterman and Gerardo Schneider

Chalmers University of Technology | University of Gothenburg



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY

## Lesson's menu

- Parallel linked queues
- Software transactional memory

# Lesson's menu

- Parallel linked queues
  - constructs and techniques
  - pick the right constructs
- Software transactional memory
  - constructs and techniques

## Learning outcomes

*Knowledge and understanding:*

- demonstrate knowledge of the issues and problems that arise in writing correct concurrent programs;
- identify the problems of synchronization typical of concurrent programs, such as race conditions and mutual exclusion

*Skills and abilities:*

- apply common patterns, such as lock, semaphores, and message-passing synchronization for solving concurrent program problems;
- apply practical knowledge of the programming constructs and techniques offered by modern concurrent programming languages;
- implement solutions using common patterns in modern programming languages

*Judgment and approach:*

- evaluate the correctness, clarity, and efficiency of different solutions to concurrent programming problems;
- judge whether a program, a library, or a data structure is safe for usage in a concurrent setting;
- pick the right language constructs for solving synchronization and communication problems between computational units.

# Synchronization costs

A number of factors **challenge** designing correct and efficient **parallelizations**:

- sequential dependencies
- synchronization costs
- spawning costs
- error proneness and composability

In **this lesson**, we present:

- a **lock-free queue** data structure, which involves minimal synchronization costs (in particular, it uses no locking)
- **software transactional memory**, which supports composability in lock-free programming

# Parallel linked queues

# Parallel linked queue

We present another example of **lock-free** data structure: an implementation of a **linked queue** that supports **parallel access**

A **queue data structure** offers obvious opportunities for **parallelization** – because insertion and removal of nodes occurs at two opposite ends of a linked structure

At the same time, it requires to **carefully consider** the interleaving of operations, and to take measures to **prevent** modifications that lead to **inconsistent** states

We will use regular Java syntax, without emphasizing opportunities for object-oriented abstraction and encapsulation, so as to have a different presentation style, complementary to the one adopted for linked sets

# The interface of a queue

We use linked lists to implement a **lock-free queue** data structures with interface:

```
interface Queue<T>
{
    // add 'item' to back of queue
    void enqueue(T item);

    // remove and return item in front of the queue
    // raise EmptyException if queue is empty
    T dequeue() throws EmptyException;
}
```

# Atomic references

To implement data structures that are correct under concurrent access without using any locks we need to rely on synchronization primitives more powerful than just reading and writing shared variables

We are going to use a variant of the compare-and-set operation:

```
class AtomicReference<V> {  
    V get();                  // current reference  
    void set(V newRef); // set reference to newRef  
    // if reference == expectRef, set to newRef and return true  
    // otherwise, do not change reference and return false  
    boolean compareAndSet(V expectRef, V newRef);  
}
```

# Nodes

The underlying implementations of queues use **singly-linked lists**, which are made of chains of nodes

- Every **node**:

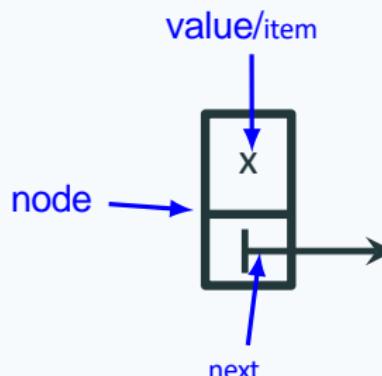
- stores an **item**— its **value**
- points to the **next** nod in the chain

To build a lock-free implementation, `next`is a reference that supports compare-and-set operations (thus, need not be **volatile**)

```
class QNode<T>
{
    // value of node
    T value;

    // next node in chain  AtomicReference<QNode<T>> next;

    QNode(T value) {
        this.value = value;
        next = new AtomicReference<>(null); }
}
```



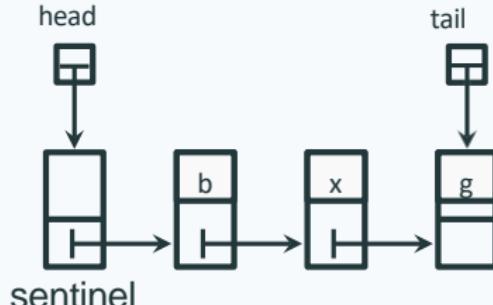
# Queues as chains of nodes

A list with a pair of **head** and **tail** references implements a queue:

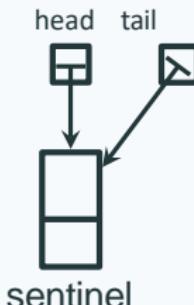
- a **sentinel** node points to the first element to be dequeued
- the queue is **empty** iff the sentinel points to **null**
- head points to the sentinel (**front** of queue)
- tail points to the latest enqueued element (**back** of queue), or the sentinel if the queue is empty

The sentinel (also called “dummy node”) ensures that head and tail are never **null**

A non-empty queue:



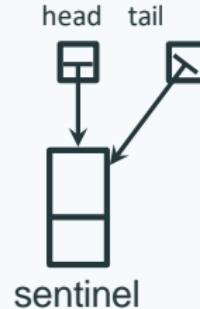
An empty queue:



# Head, tail, and empty queue

```
class LockFreeQueue<T> implements Queue<T>
{
    // access to front and back of queue
    protected AtomicReference<QNode<T>> head;
    protected AtomicReference<QNode<T>> tail;

    // constructor creating empty queue
    public LockFreeQueue() {
        // value of sentinel does not matter
        QNode<T> sentinel = new QNode<>();
        head = new AtomicReference<>(sentinel);
        tail = new AtomicReference<>(sentinel);
    }
}
```



## Enqueue operation

The method **enqueue adds** a new node to the **back of a queue** – where tail points. It requires two updates that modify the linked structure:

1. **update last:** make the last node in the queue point to the new node
2. **update tail:** make tail point to the new node

Each update is individually **atomic** (it uses compare-and-set), but another thread may interfere between the two updates:

- **repeat** update last until success
- try update tail **once**
- the implementation should be able to deal with a “half finished” enqueue operation (tail not updated yet), and **finish the job** – this technique is called **helping**

# Method enqueue

```
public void enqueue(T value) {  
    // new node to be enqueued  
    QNode<T> node = new QNode<>(value);  
    while (true) { // nodes at back of queue  
        QNode<T> last = tail.get();  
        QNode<T> nextToLast = last.next.get();  
        // if tail points to last  
        if (last == tail.get())  
        { // and if last really has no successor  
            if (nextToLast == null) {  
                // make last point to new node  
                if (last.next.compareAndSet(nextToLast, node))  
                    // if last.next updated, try once to update tail  
                    { tail.compareAndSet(last, node); return; }  
            } else // last has valid successor: try to update tail and repeat  
            { tail.compareAndSet(last, nextToLast); } } }  
}  
fails only if another thread moves tail
```

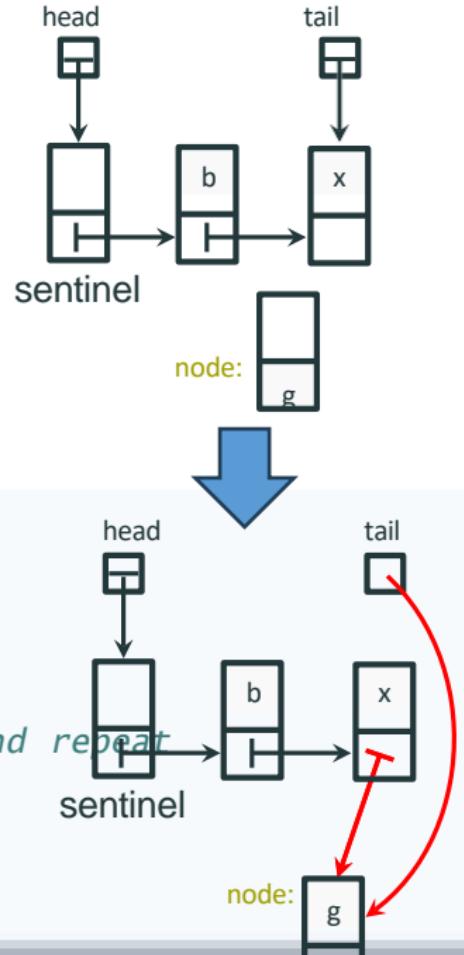
helps another thread move tail

# Method enqueue

```

public void enqueue(T value) {
  // new node to be enqueued
  QNode<T> node = new QNode<>(value);
  while (true) { // nodes at back of queue
    QNode<T> last = tail.get();
    QNode<T> nextToLast = last.next.get();
    // if tail points to last
    if (last == tail.get())
    { // and if last really has no successor
      if (nextToLast == null) {
        // make last point to new node
        if (last.next.compareAndSet(nextToLast, node))
          // if last.next updated, try once to update tail
          { tail.compareAndSet(last, node); return; }
      } else // last has valid successor: try to update tail and repeat
      { tail.compareAndSet(last, nextToLast); } } }
}
  
```

If tail points to actual last:

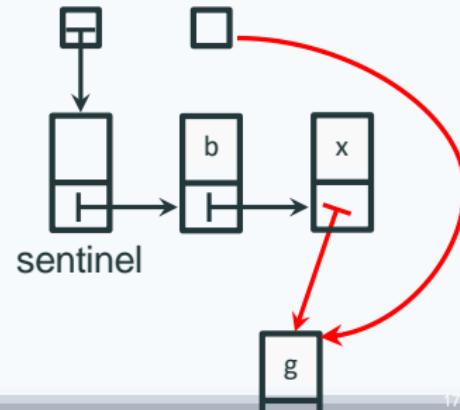
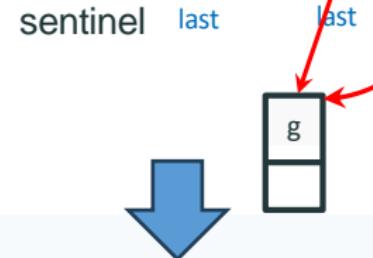
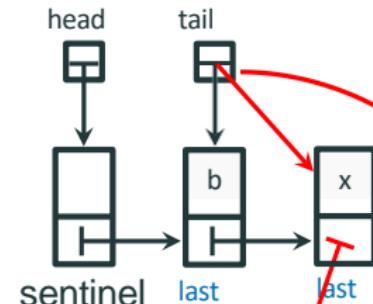


# Method enqueue

```

public void enqueue(T value) {
  // new node to be enqueued
  QNode<T> node = new QNode<>(value);
  while (true) { // nodes at back of queue
    QNode<T> last = tail.get();
    QNode<T> nextToLast = last.next.get();
    // if tail points to last
    if (last == tail.get())
    { // and if last really has no successor
      if (nextToLast == null) {
        // make last point to new node
        if (last.next.compareAndSet(nextToLast, node))
          // if last.next updated, try once to update tail
          { tail.compareAndSet(last, node); return; }
      } else // last has valid successor: try to update tail
        and repeat
        { tail.compareAndSet(last, nextToLast); } } }
}
  
```

If tail points to old last:



## Dequeue operation

The method **dequeue removes** the node at the **head of a queue** (where the **sentinel** points)

Unlike enqueue, **dequeueing** only requires one update to the linked structure:

- **update head:** make head point the node previously pointed to by the sentinel; the same node becomes the new sentinel and is also returned

The update is **atomic** (it uses compare-and-set), but other threads may be updating the head concurrently:

- **repeat** update head until success
- if you detect a “half finished” enqueue operation – with the tail pointing to the sentinel about to be removed – **help** by moving the tail forward

# Method dequeue

```
public T dequeue() throws EmptyException {
    while (true) // nodes at front, back of queue
    { QNode<T> sentinel = head.get(),
        last = tail.get(),
        first = sentinel.next.get();
        if (sentinel == head.get()) // if head points to sentinel
        { // if tail also points to sentinel
            if (sentinel == last)
            { // empty queue: raise exception
                if (first == null)
                    throw new EmptyException();
                // non-empty: update tail, repeat
                tail.compareAndSet(last, first); ← must help move tail before updating head
            }
            else // tail doesn't point to sentinel
            { T value = first.value;
                // make head point to first (new sentinel); retry until success
                if (head.compareAndSet(sentinel, first)) return value; } } }
    }
```

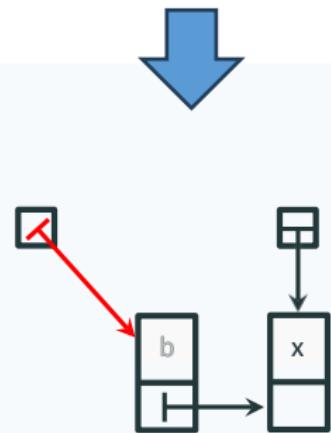
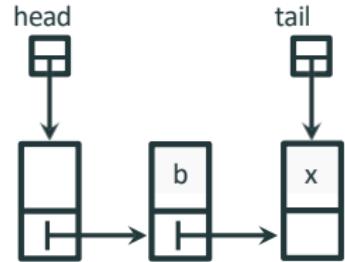
must move head: no other thread can help

# Method dequeue

```

public T dequeue() throws EmptyException {
  while (true) // nodes at front, back of queue
  { QNode<T> sentinel = head.get(),
    last = tail.get(),
    first = sentinel.next.get();
    if (sentinel == head.get()) // if head points to sentinel
    { // if tail also points to sentinel
      if (sentinel == last)
      { // empty queue: raise exception
        if (first == null)
          throw new EmptyException();
        // non-empty: update tail, repeat
        tail.compareAndSet(last, first);
      }
      else // tail doesn't point to sentinel
      { T value = first.value;
        // make head point to first (new sentinel); retry until success
        if (head.compareAndSet(sentinel, first)) return value; } } }
}
  
```

If tail needs no update:

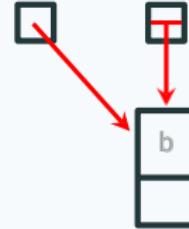
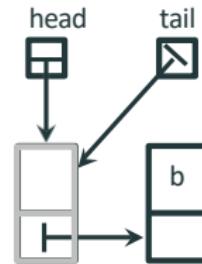


# Method dequeue

```

public T dequeue() throws EmptyException {
  while (true) // nodes at front, back of queue
  { QNode<T> sentinel = head.get(),
    last = tail.get(),
    first = sentinel.next.get();
    if (sentinel == head.get()) // if head points to sentinel
    { // if tail also points to sentinel
      if (sentinel == last)
      { // empty queue: raise exception
        if (first == null)
          throw new EmptyException();
        // non-empty: update tail, repeat
        tail.compareAndSet(last, first);
      }
      else // tail doesn't point to sentinel
      { T value = first.value;
        // make head point to first (new sentinel); retry until success
        if (head.compareAndSet(sentinel, first)) return value; } } }
}
  
```

If tail needs update:



# Garbage collection saves the day

If we were using a language **without garbage collection** – where objects can be recycled – the following **problem** could occur:

1. *t* is about to CAS head from sentinel node a to node b:

```
head.compareAndSet(sentinel, first)
```

2. *u* dequeues b and x
3. *u* enqueues a again (**the very same node**), enqueues y, enqueues p, and then dequeues a again, so that the same node a becomes the sentinel again
4. *t* completes CAS successfully (head still points to *t*'s local reference *sentinel*), but node b is now disconnected!

# The ABA problem

The problem we have just seen is known as the **ABA problem**

It cannot occur in languages that, like Java, feature **automatic memory management (garbage collection)**



Our LockFreeQueue implementation **relies on garbage collection for correctness**: a thread creates a **fresh node** (using **new**) whenever it enqueues a value, which is guaranteed to have a reference that was not in use before

# Software Transactional Memory

# Challenges to parallelization

A strategy to **parallelize** a task ( $F, D$ ) should be:

- **correct**: the overall result of the parallelization is  $F(D)$
- **efficient**: the total resources (time and memory) used to compute the parallelization are less than those necessary to compute ( $F, D$ ) sequentially

A number of factors **challenge** designing correct and efficient **parallelizations**:

- sequential dependencies – seen
- synchronization costs – seen
- spawning costs – seen
- **error proneness and compositability**

# Error proneness and composability

Synchronization is prone to errors such as **data races**, **deadlocks**, and **starvation**

From the point of view of software construction, the lack of **composability** is a challenge that prevents us from developing parallelization strategies that are **generally applicable**

# Error proneness and composability

Consider an `Account` class with methods `deposit` and `withdraw` that execute atomically  
What happens if we combine the two methods to implement a `transfer` operation?

```
class Account {  
    synchronized void  
    deposit(int amount)  
    { balance += amount; }  
  
    synchronized void  
    withdraw(int amount)  
    { balance -= amount; }  
}
```

execute uninterruptedly

```
class TransferAccount  
    extends Account {  
  
    // transfer from 'this' to 'other'  
    void transfer(int amount, Account other)  
    { this.withdraw(amount);  
        other.deposit(amount); }  
}
```

Method `transfer` does **not** execute **uninterruptedly**: other threads can execute between the call to `withdraw` and the call to `deposit`, possibly preventing the transfer from succeeding

(For example, `Account other` may be closed; or the total balance temporarily looks lower than it is!)

# Composability

```
class Account {  
    void // thread unsafe!  
    deposit(int amount)  
    { balance += amount; }  
    void // thread unsafe!  
    withdraw(int amount)  
    { balance -= amount; }  
}  
  
class TransferAccount  
extends Account {  
    // transfer from 'this' to 'other'  
    synchronized void  
    transfer(int amount, Account other)  
    { this.withdraw(amount);  
        other.deposit(amount); }  
}
```

None of the [natural solutions to composing](#) is fully satisfactory:

- let clients of `Account` do the locking where needed – error proneness, revealing implementation details, scalability
- recursive locking – risk of deadlock, performance overhead

With [message passing](#), we encounter similar problems – synchronizing the effects of messaging two independent processes

# Transactions

The notion of transaction, which comes from database research, supports a general approach to lock-free programming:

A **transaction** is a **sequence** of steps executed by a single thread, which are executed **atomically**

A transaction may:

- **succeed**: all changes made by the transaction are **committed** to shared memory; they appear as if they happened instantaneously
- **fail**: the partial changes are **rolled back**, and the shared memory is in the same state it would be if the transaction had never executed

Therefore, a transaction either executes completely and successfully, or it does not have any effect at all

# Programming with transactions

The notion of transaction supports a **general approach** to **lock-free** programming:

- define a transaction for every access to shared memory
- if the transaction succeeds, there was no interference
- if the transaction failed, **retry** until it succeeds

Imagine we have a syntactic means of defining **transaction code**:

```
atomic {                      % execute Function(Arguments)
    // transaction code      % as a transaction (retry until success)
}
        atomic(Function, Arguments)
// retry until success
```

Transactions may also support invoking **retry** and **rollback** explicitly

(Note that **atomic** is not a valid keyword in Java or Erlang: we use it for illustration purposes, and later we sketch how it could be implemented as a function in Erlang)

# Transactions are better than locks

Transactional atomic blocks look superficially similar to monitor's methods with implicit locking, but they are in fact much **more flexible**:

- since transactions do not lock, there is **no** locking **overhead**
- **parallelism** is achieved without risks of race conditions
- since no locks are acquired, there is **no** problem of **deadlocks** (although starvation may still occur if there is a lot of contention)
- transactions **compose** easily

```
class Account {
    void deposit(int amount)
    { atomic {
        balance += amount; }
    void withdraw(int amount)
    { atomic {
        balance -= amount; }
    }
}
```

```
class TransferAccount extends Account {
    // transfer from 'this' to 'other'
    void transfer(int amount,
                  Account other)
    { atomic {
        this.withdraw(amount);
        other.deposit(amount); }
    }
}
```

no locking, so no deadlock is possible!

# Transactional memory

A **transactional memory** is a shared memory storage that supports atomic updates of **multiple memory locations**

**Implementations** of transactional memory can be based on hardware or software:

- **hardware** transactional memory relies on support at the level of instruction sets (Herlihy & Moss, 1993)
- **software** transactional memory is implemented as a library or language extension (Shavit & Touitou, 1995)

Software transactional memory implementations are available for several mainstream languages (including Java, Haskell, and Erlang)

This is still an active research topic – quality varies!

# Implementing software transactional memory

We outline an implementation of software transactional memory (**STM**) in [Erlang](#)

Each variable in an STM is identified by a name, value, and [version](#):

```
-record(var, {name, version = 0, value = undefined}).
```

Clients use an STM as follows:

- at the beginning of a transaction, [check out](#) a copy of all variables involved in the transaction
- execute the transaction, which modifies the [values](#) of the [local](#) copies of the variables
- at the end of a transaction, try to [commit](#) all local copies of the variables

# Implementing software transactional memory

We outline an implementation of software transactional memory (**STM**) in [Erlang](#)

Each variable in an STM is identified by a name, value, and [version](#):

```
-record(var, {name, version = 0, value = undefined}).
```

The STM's [commit](#) operation ensures atomicity:

- if all committed variables have the [same version number](#) as the corresponding variables in the STM, there were [no changes](#) to the memory during the transaction: the transaction [succeeds](#)
- if some committed variable has a [different version number](#) from the corresponding variable in the STM, there was [some change](#) to the memory during the transaction: the transaction [fails](#)

# The counter example – with software transactional memory

```
int cnt;
```

---

thread t

```
int c;  
atomic {  
    c = cnt;  
    cnt = c + 1;  
}
```

thread u

```
int c;  
atomic {  
    c = cnt;  
    cnt = c + 1;  
}
```

The **atomic** translates into a **loop** that repeats **until** the transaction **succeeds**:

1. check out (pull) the current value of cnt
2. increment the local variable c
3. try to commit (push) the new value of cnt
4. if cnt has changed version when trying to commit, repeat the loop

## Atomic -&gt; do-while

---

(name: cnt, version:x, value:y)

---

thread t

```
int c;  
do {  
    // check out cnt  
    c = pull(cnt);  
    c = c + 1;  
} while (!push(cnt, c));  
// commit cnt
```

thread u

```
int c;  
do {  
    // check out cnt  
    c = pull(cnt);  
    c = c + 1;  
} while (!push(cnt, c));  
// commit cnt
```

# STM in Erlang

An **STM** is a **server** that provides the following main operations:

- **pull(Name)**: check out a copy of variable with name **Name**
- **push(Vars)**: commit all variables in **Vars**; return **fail** if unsuccessful

Clients read and write **local copies** of variables using:

- **read(Var)**: get value of variable **Var**
- **write(Var, Value)**: set value of variable **Var** to **Value**

We base the STM implementation on the **gserver** generic server implementation we presented in a previous lectures.

## STM: operations

```
create(Tm, Name, value) ->
  gserver:request(Tm, {create, Name, value}).  
  
drop(Tm, Name) ->
  gserver:request(Tm, {drop, Name}).  
  
pull(Tm, Name) ->
  gserver:request(Tm, {pull, Name}).  
  
push(Tm, Vars) when is_list(Vars) ->
  gserver:request(Tm, {push, Vars});  
  
read(#var{value = Value}) -> Value.  
write(Var = #var{}, Value) -> Var#var{value = Value}.
```

# STM: server handlers

The storage is a **dictionary** associating variable names to variables; it is the essential part of the server state

```
stm(Storage, {pull, Name}) ->
  case dict:is_key(Name, Storage) of
    true ->
      {reply, Storage,
       dict:fetch(Name, Storage)};
    false ->
      {reply, Storage, not_found}
  end;
```

```
stm(Storage, {push, Vars}) ->
  case try_push(Vars, Storage) of
    {success, NewStorage} ->
      {reply, NewStorage, success};
    fail ->
      {reply, Storage, fail}
  end.
```

## STM: try to push

The helper function `try-push` determines if any variable to be committed has a different version from the corresponding one in the STM

```
try_push([], Storage) ->
  {success, Storage};
try_push([Var = #var{name = Name, version = Version} | Vars], Storage) ->
  case dict:find(Name, Storage) of
    {ok, #var{version = Version}} ->
      try_push(Vars,
              dict:store(Name, Var#var{version = Version + 1},
              Storage));
    _ -> fail
  end.
```

# Using the Erlang STM

Using the STM to create **atomic functions** is quite straightforward

An atomic **pop** operation for a list:

```
% pop head element from 'Name'
qpop(Tm, Name) ->
  Queue = pull(Tm, Name),
  [H|T] = read(Queue),
  NewQueue = write(Queue, T),
  case push(Tm, NewQueue) of
    % push failed: retry!
    fail -> qpop(Tm, Name);
    % push successful: return head
    _ -> H
  end.
```

An atomic **push** operation for a list:

```
% push 'Value' to back of 'Name'
qpush(Tm, Name, Value) ->
  Queue = pull(Tm, Name),
  vals = read(Queue),
  NewQueue = write(Queue,
                    vals ++ [value]),
  case push(Tm, NewQueue) of
    % push failed: retry!
    fail -> qpush(Tm, Name, Value);
    % push successful: return ok
    _ -> ok
  end.
```

# Composable transactions?

The simple implementation of STM we have outlined does not support easily **composing** transactions:

```
% pop from Queue1 and push to Queue2
qtransfer(Tm, Queue1, Queue2) ->
    value = qpop(Tm, Queue1),   % another process may interfere!
    qpush(Tm, Queue2, value).
```

To implement composability, we need to keep track of **pending transactions** and defer commits until all nested transactions are done

See the course's website for an example implementation:

```
% atomically execute Function on arguments Args
atomic(Tm, Function, Args) -> todo.
```

# These slides' license

© 2016–2019 Carlo A. Furia, Sandro Stucki



Except where otherwise noted, this work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

To view a copy of this license, visit

<http://creativecommons.org/licenses/by-sa/4.0/>.

# Weak Memory Models

Lecture X of TDA384/DIT391

Principles of Concurrent Programming

Nir Piterman and Gerardo Schneider

Chalmers University of Technology | University of Gothenburg

Based on material prepared by Andreas Lööw



UNIVERSITY OF  
GOTHENBURG



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY

# Telling the truth

- Why synchronization?
  - Atomicity!
  - Visibility!
- We have used modelling languages and pseudo-code.
- Real languages (e.g., Java) have additional issues:
  - Memory model – how threads interact through memory and share data.
- In this lecture:
  - Rudiments of the Java Memory Model and how to program in it.
  - Principles applying to concurrent programming in other languages.

# Telling th

## Instruction execution order

- Why synch
    - Atomicity
    - Visibility
  - We have used threads
  - Real languages
    - Memory
  - In this lecture
    - Rudimentary
    - Principles
- When we designed and analyzed concurrent algorithms, we implicitly assumed that threads execute instructions in textual program order  
This is **not guaranteed** by the Java language – or, for that matter, by most programming languages – when threads access shared fields  
(Read “The silently shifting semicolon” <http://drops.dagstuhl.de/opus/volltexte/2015/5025/> for a nice description of the problems)
- Compilers may reorder instructions based on static analysis, which does not know about threads.
  - Processors may delay the effect of writes to memory
- This adds to the complications of writing low-level concurrent software correctly



# Lesson's menu

- What are memory models?
- Why weak memory models?
- Something about the Java Memory Model (as an example of a weak memory model)
- Programming in the JMM

# What are memory models?

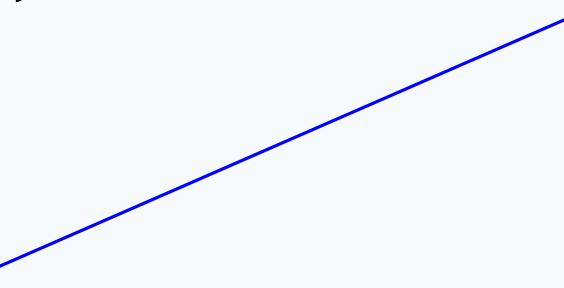
# Memory Models

- As part of language semantics:
  - How threads communicate through shared memory.
  - What values are variable reads allowed to return?
- There are different memory models:
  - **Sequential Consistency** – one of the “strongest” memory models. Often assumed for pseudocode (and up to now in this course).
  - Java uses **Java Memory Model (JMM)** – a weak memory model.

# Reading variables: Sequential programming

```
int x = 0;  
int y = 0;  
x = 1;  
y = 1;  
print(y);  
print(x);
```

What value will this read of y return?  
Obviously 1! We always get the latest value!



# Reading variables: Concurrent programming

```
bool done = false; int res = 0;
```

```
green_thread {  
1 res = 666;  
2 done = true;  
3}
```

```
blue_thread {  
1 if (done)  
2   print(res);  
3}
```

What are the possible outcomes of running?  
Let's consider all possible interleavings.

# Reading variables: Concurrent programming

```
bool done = false;  
int res = 0;  
green_thread {  
1 res = 666;  
2 done = true;  
3}
```

1;1;2;

(x)= Variables	
	Breakpoints
	Expressions
res	666
done	true

No output

1;1;2;

(x)= Variables	
	Breakpoints
	Expressions
res	
done	true

No output

```
blue_thread {  
1 if (done)  
2 print(res);  
3}
```

1;2;1;2;

(x)= Variables	
	Breakpoints
	Expressions
res	666
done	true

Output 666

Let's see what Java says ...

Demo OutOfOrderTest.java

# Reading variables: Sequential consistency (SC)

Some visibility guarantees in SC:

- “Program order” always maintained
  - In particular, `r = 666` always before `done= true` in any interleaving
- No “stale” values: Always see the latest value written to any variable

But the above guarantees not provided by all weak memory models (e.g. JMM)!

Interleaving-based semantics is the “obvious” semantics.

Why make things more difficult? Why give up program order?

Because sequential consistency costs too much.

```
bool done = false;  
int res = 0;  
green_thread {  
    res = 666;  
    done = true;  
}  
  
blue_thread {  
    if (done)  
        print(res);  
}
```

# Take home message 1

You must understand the memory model in order to write correct programs.

# Why weak-memory models?

# SC problem 1: Compiler optimizations

For some compiler optimizations we want to reorder writes to variables.

This does not happen in pseudocode ...

Messy details ...

# SC problem 1: Compiler optimizations

- E.g., the transformation to the right “semantics preserving” in sequential setting if we only consider final state of program
- Not equivalent if we can inspect program under execution, which we can if  $x$  and  $y$  are shared variables in a concurrent setting
- Breaks illusion of “program order”!

Write order swapped

Original program:

```
x = 1;  
y = 2;  
z = x + y; // x = 1, y = 2,  
z = 3
```

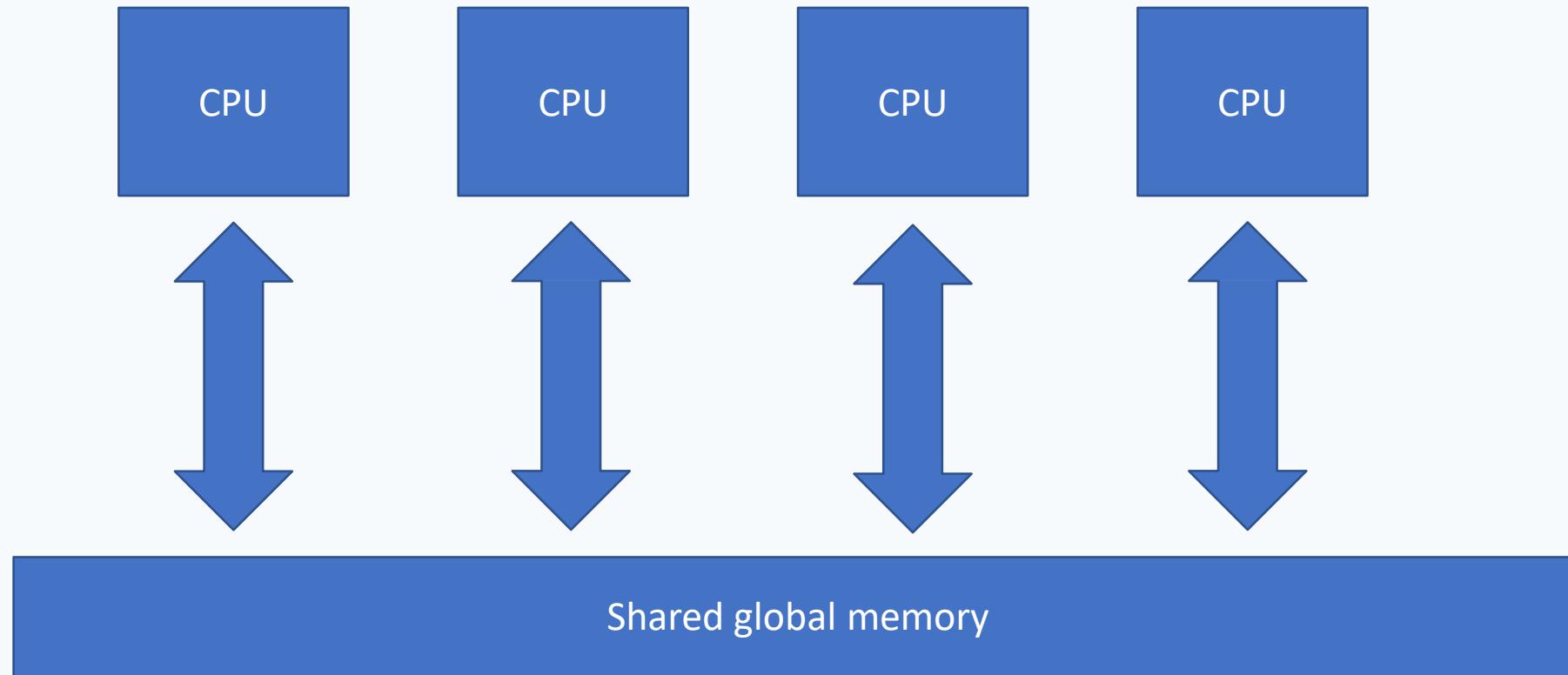
Transformed program:

```
y = 2;  
x = 1;  
z = x + y; // x = 1, y = 2,  
z = 3
```

Write order  
swapped

# SC cost 2: Causes too much cache synchronization

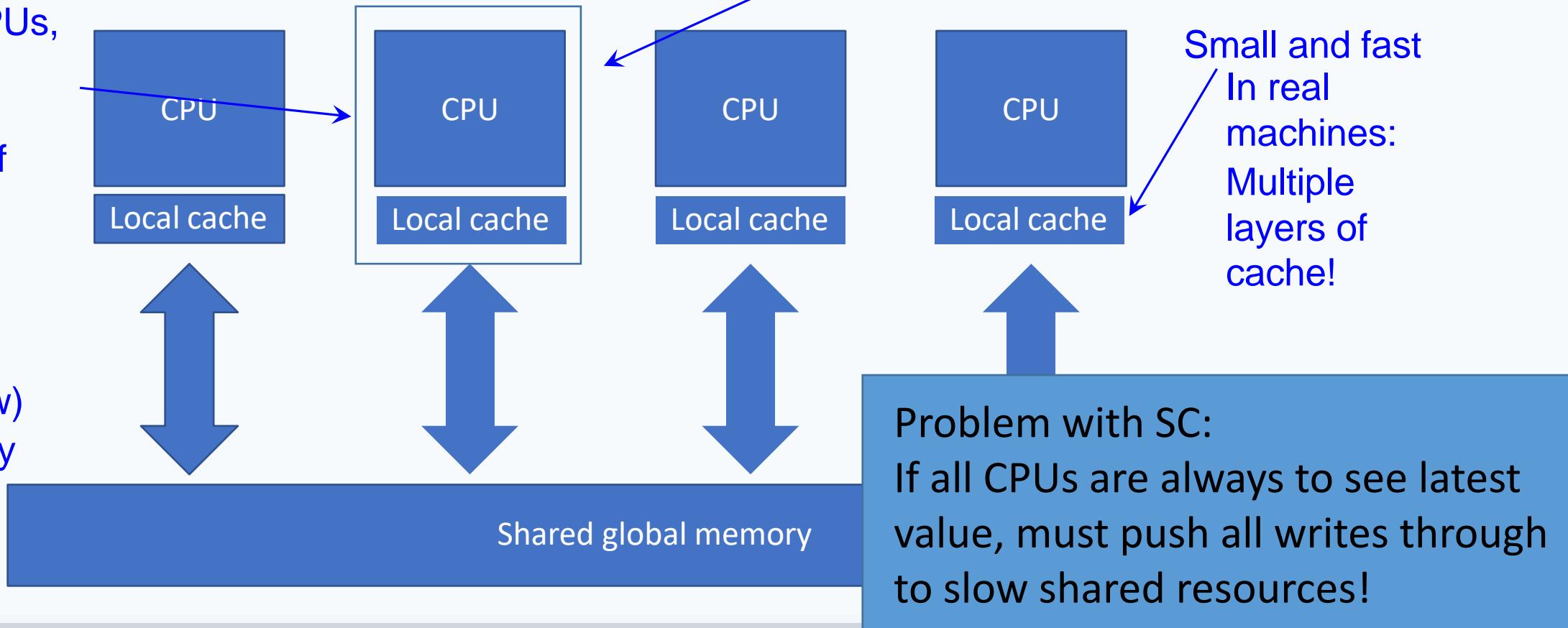
Cost of SC not obvious with too simplified machine models:



# SC cost 2: Causes too much cache

Slightly more realistic model of today's computers:

In modern CPUs, even a single CPU may execute out of order and in parallel ...



# Why not SC?

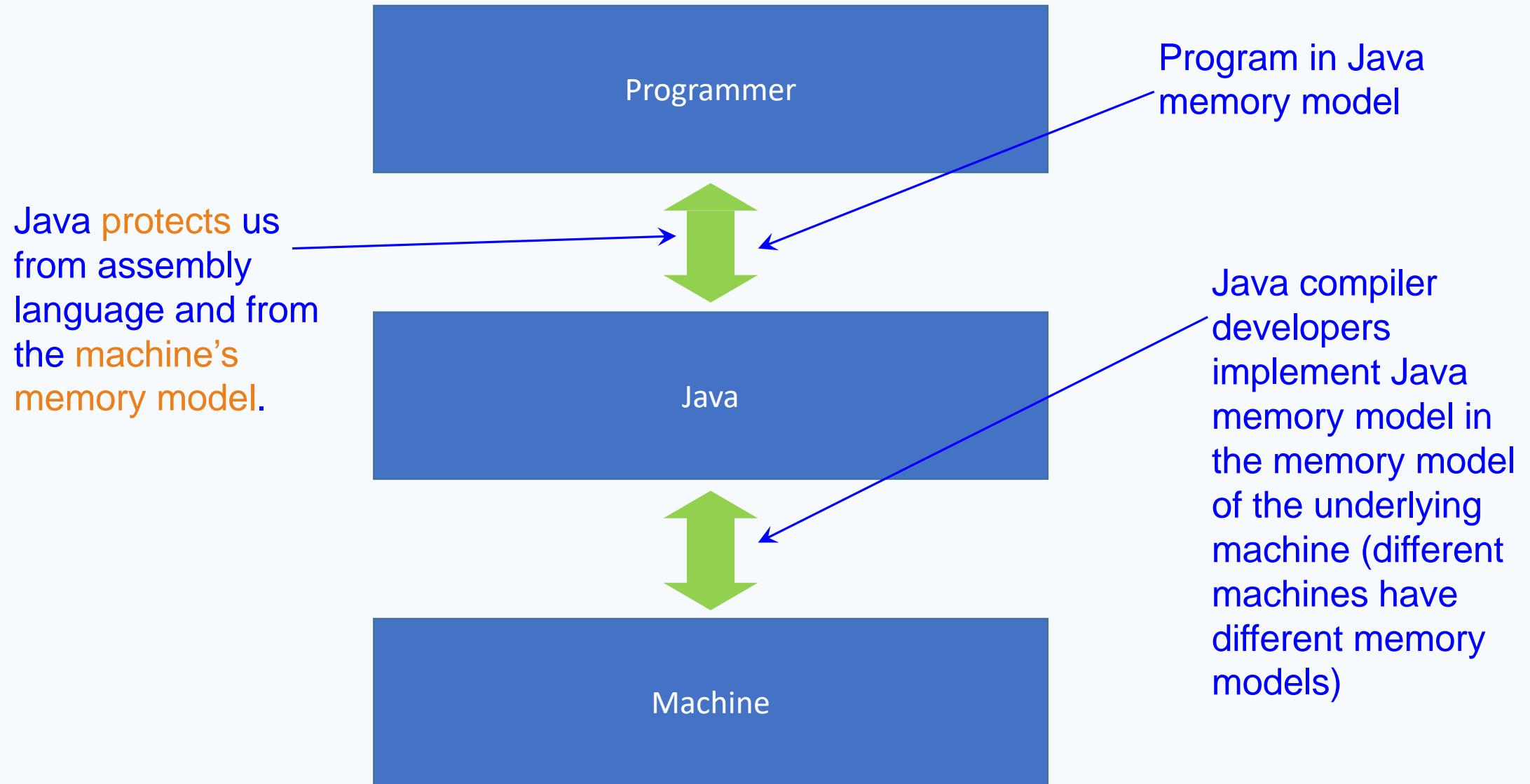
- Examples:
  - Out of order execution
  - Compiler optimizations
  - Avoid communication
- SC too expensive in many situations
- Solution to mentioned problems:  
Relax some guarantees offered by SC → we get weak memory models

Weaker memory models (potentially) more performant, but more difficult to program in

# Something about JMM

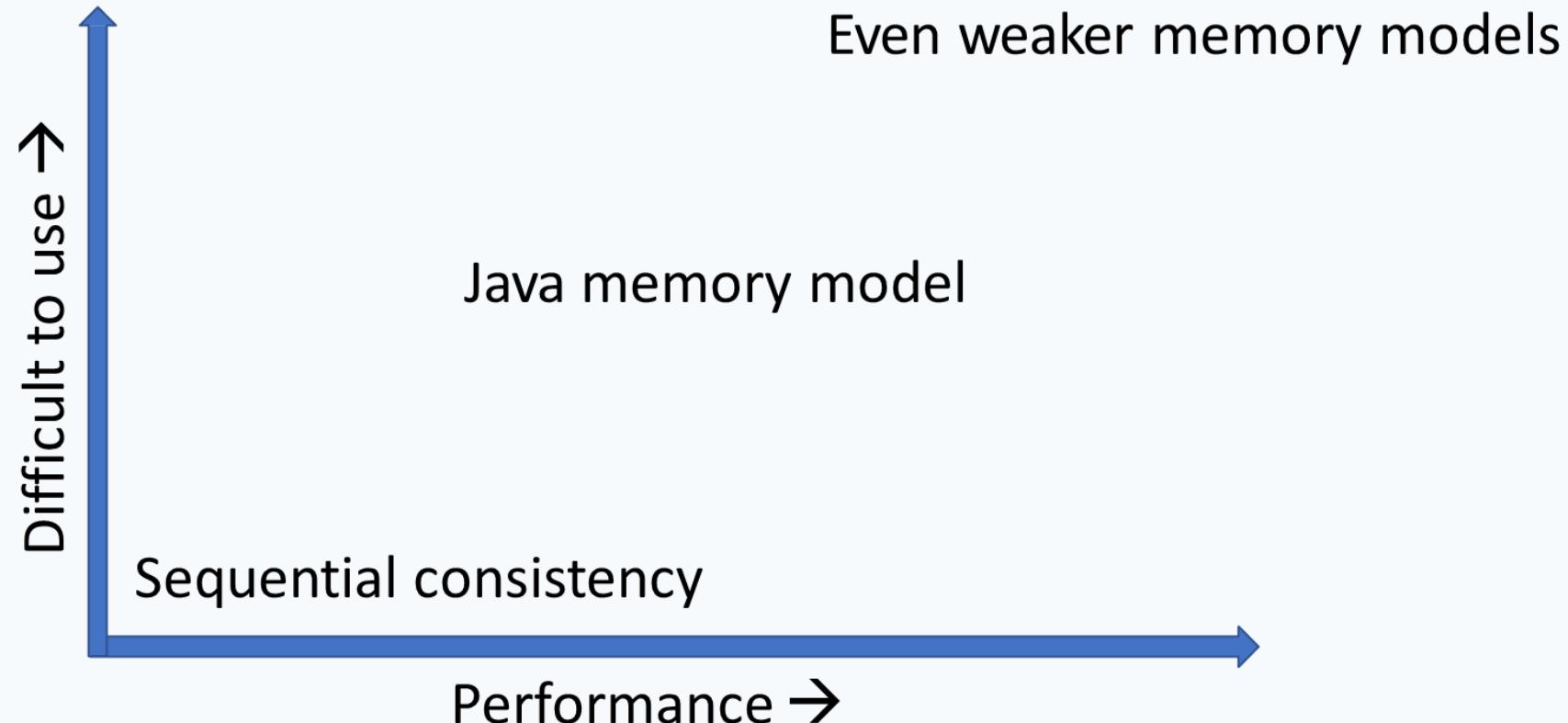
Example of a weak memory model

# More context: machine details



# The Java memory model

- Less convenient than SC, but implementable on modern machine architectures without too much performance loss
- There is no “right design”:



# SC for data-race-free programs

- A few languages have converged to “sequential consistency for data-race-free programs” memory models
- Java included in this family
- Reasoning principle: If there are no data races (under SC), we can assume SC when reasoning about our program
- Important to remember definitions of data race and race conditions

Data races

Race conditions are typically caused by a lack of synchronization between threads that access shared memory

A data race occurs when two concurrent threads:

- Access a shared memory location
- At least one access is a write
- The threads use no explicit synchronization mechanism to protect the shared data

30

Race conditions

Concurrent programs are nondeterministic:

- Executing multiple times the same concurrent program with the same inputs may lead to different execution traces
- A result of the nondeterministic interleaving of each thread's trace to determine the overall program trace
- In turn, the interleaving is a result of the scheduler's decisions

A race condition is a situation where the correctness of a concurrent program depends on the specific execution

The concurrent counter example has a race condition:

- in some executions the final value of counter is 2 (correct)
- in some executions the final value of counter is 1 (wrong)

Race conditions can greatly complicate debugging!

28

# Data races: slight (Java) variation

Def.

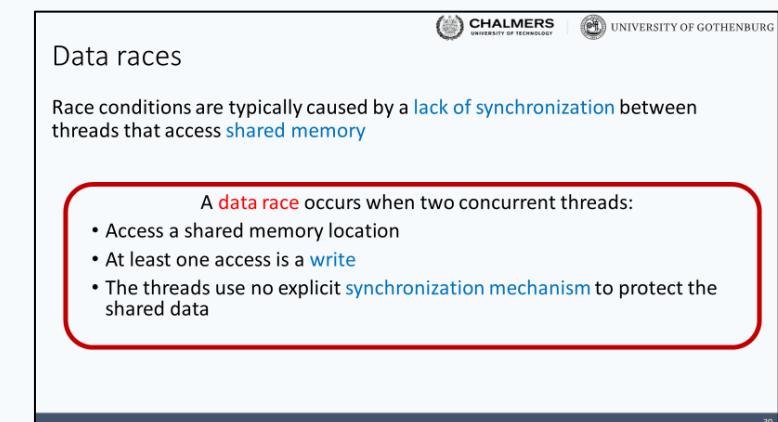
Two memory accesses are in a **data race** iff they access the same **memory location simultaneously** (they are interleaved next to each other), at least one access is a **write**, **insufficient explicit synchronization** used to protect the accesses

Def.

A program is **data-race-free** iff no SC execution of the program contains a data race

Notes:

- We quantify over all SC executions in the second
- Data-race-freedom is a “**language-level**” property!



CHALMERS  
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

## Data races

Race conditions are typically caused by a lack of synchronization between threads that access shared memory

A **data race** occurs when two concurrent threads:

- Access a shared memory location
- At least one access is a **write**
- The threads use no explicit synchronization mechanism to protect the shared data

# Definition of data race surprisingly subtle

Does this program contain any data races?

```
bool x = false, y = false;
```

```
t1 {  
    if (x) y = true;  
}
```

```
t2 {  
    if (y) x = true;  
}
```

# Race conditions

## Race conditions

Concurrent programs are **nondeterministic**:

- Executing multiple times the same concurrent program with the same inputs may lead to **different execution traces**
- A result of the nondeterministic **interleaving** of each thread's trace to determine the overall program trace
- In turn, the interleaving is a result of the **scheduler**'s decisions

A **race condition** is a situation where the correctness of a concurrent program depends on the specific execution

The **concurrent counter** example has a **race condition**:

- in some executions the final value of counter is **2** (correct)
- in some executions the final value of counter is **1** (wrong)

Race conditions can greatly **complicate debugging!**

28

Note that this is an “**application-level**” property!

I.e., for a given program p, to answer the question “is p free from race conditions?” we must have access to the specification of p.

# SC for data-race-free programs, again

- For Java programs, we have SC for programs **without data races**
- Reasoning principle in more detail:
  1. Assume SC and make sure that there are no data races
  2. If no data races, we can assume SC when reasoning about race conditions
- What about the semantics of programs *with* data races?
  - Will not be considered here
  - In e.g. C++ data races result in undefined behavior (see C++ specification or [https://en.cppreference.com/w/cpp/language/memory\\_model](https://en.cppreference.com/w/cpp/language/memory_model))
  - Java is supposed to be a "safe language", some guarantees

# Programming in the JMM

As an example of a weak memory model

# What does all this mean in practice?

- I.e: How does “weak memory models” affect *my daily life as a programmer?*
- Answer: You must “**annotate**” your program more than with SC
  - Sprinkle additional synchronization information on top of your program
  - Variable qualifiers, synchronization mechanisms (e.g. locks), etc.
  - Exactly what “annotate” means **depends on language**
- Essentially, you annotate which data/actions are shared and which are not

# Simpler example: only one variable!

```
bool done = false;
```

```
t1 {  
    done = true;  
}
```

```
t2 {  
    if (done) print(33);  
}
```

- Does this program contain
  - data races?
  - race conditions?
- Data race = yes, done is accessed without synchronization and one of the accesses is a write
- Race condition = depends on the specification we are to satisfy (what it means for the program to be correct)
- Race condition = even if we had a specification, we have a data race so our reasoning principle does not apply!

- There is a problem with this program!
- From SC perspective, everything is fine!
- No atomicity problems ... but visibility problems!

# Simple example (fixed)

```
volatile done = false;
```

```
t1 {  
    done = true;  
}
```

```
t2 {  
    if (done) print(33);  
}
```

- Solution: Annotate your program. E.g., in Java `volatile` is considered synchronization.
- Does this program contain
  - data races?
  - race conditions?
- Data race = no, in Java `volatile` accesses are considered synchronized
- Race condition = still depends on specification
- Example spec: “If the program outputs something, it must output 33”.
- Race condition = no, for the above specification the correct output does not depend on specific execution/interleaving.
- Example spec: “The program outputs 33”.
- Race condition = yes, some interleavings give us the correct output, others do not.

# Similar example, with locks

```
lock lock = new Lock();
int id = 0;
```

```
t1 {
    lock.lock();
    id++;
    lock.unlock();
}
```

```
t2 {
    print(id);
}
```

Data races?

We have a race! All accesses to the shared variable done must be synchronized!

Here we have (again) atomicity, but not **visibility**

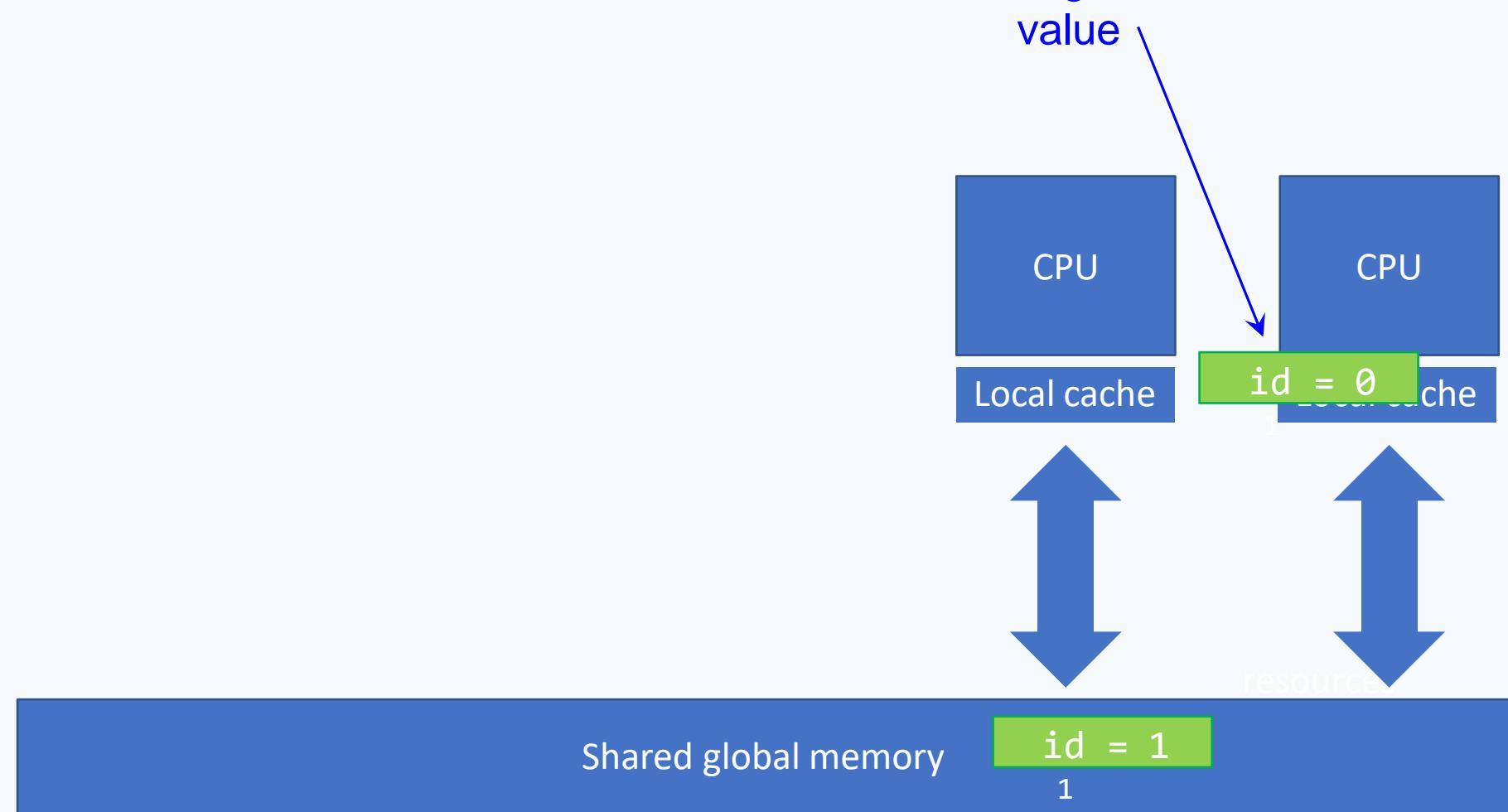
# id might exist as multiple copies...

```
lock lock = new lock();  
int id = 0;
```

```
t1 {  
    lock.lock();  
    id++;  
    lock.unlock();  
}
```

```
t2 {  
    print(id);  
}
```

Might read “stale” value



# Similar example, with locks (fixed)

```
lock lock = new Lock();
int id = 0;
```

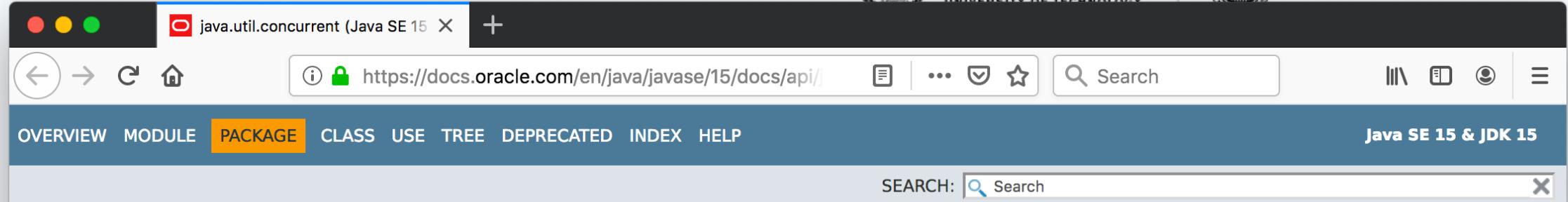
This is how the program would look like  
with proper annotations/synchronization

```
t1 {
    lock.lock();
    id++;
    lock.unlock();
}
```

Now there are no data races.

```
t2 {
    lock.lock(); // new
    print(id);
    lock.unlock(); // new
}
```

# JMM in More Detail



A screenshot of a web browser window displaying the Java.util.concurrent package documentation. The browser has three tabs open: 'java.util.concurrent (Java SE 15)' (the active tab), 'java.util.concurrent.ForkJoinPool (Java SE 15)', and '+'. The address bar shows the URL: https://docs.oracle.com/en/java/javase/15/docs/api/. The page header includes links for OVERVIEW, MODULE, PACKAGE (highlighted in orange), CLASS, USE, TREE, DEPRECATED, INDEX, and HELP, along with a 'Java SE 15 & JDK 15' link. A search bar at the top right contains the text 'Search'.

## Module java.base

# Package java.util.concurrent

Utility classes commonly useful in concurrent programming. This package includes a few small standardized extensible frameworks, as well as some classes that provide useful functionality and are otherwise tedious or difficult to implement. Here are brief descriptions of the main components. See also the [java.util.concurrent.locks](#) and [java.util.concurrent.atomic](#) packages.

## Executors

**Interfaces.** [Executor](#) is a simple standardized interface for defining custom thread-like subsystems, including thread pools, asynchronous I/O, and lightweight task frameworks. Depending on which concrete Executor class is being used, tasks may execute in a newly created thread, an existing task-execution thread, or the thread calling [execute](#), and may execute sequentially or concurrently. [ExecutorService](#) provides a more complete asynchronous task execution framework. An ExecutorService manages queuing and scheduling of tasks, and allows controlled shutdown. The [ScheduledExecutorService](#) subinterface and associated interfaces add support for delayed and periodic task execution. ExecutorServices provide methods arranging asynchronous execution of any function expressed as [Callable](#), the result-bearing analog of [Runnable](#). A [Future](#) returns the results of a function, allows determination of whether execution has completed, and provides a means to cancel execution. A [RunnableFuture](#) is a Future that possesses a [run](#) method that upon execution, sets its results.

**Implementations.** Classes [ThreadPoolExecutor](#) and [ScheduledThreadPoolExecutor](#) provide tunable, flexible thread pools. The [Executors](#) class provides factory methods for the most common kinds and configurations of Executors, as well as a few utility methods for using them. Other utilities based on Executors include the concrete class [FutureTask](#) providing a common extensible implementation of Futures, and [ExecutorCompletionService](#), that assists in coordinating the processing of groups of asynchronous tasks.

Class [ForkJoinPool](#) provides an Executor primarily designed for processing instances of [ForkJoinTask](#) and its subclasses. These classes employ a work-stealing scheduler that attains high throughput for tasks conforming to restrictions that often hold in computation-intensive parallel processing.

## Queues

SEARCH:  Search X

- they are guaranteed to traverse elements as they existed upon construction exactly once, and may (but are not guaranteed to) reflect any modifications subsequent to construction.

## Memory Consistency Properties

### Or memory consistency model

Chapter 17 of *The Java Language Specification* defines the *happens-before* relation on memory operations such as reads and writes of shared variables. The results of a write by one thread are guaranteed to be *visible* to a read by another thread only if the write operation *happens-before* the read operation. The `synchronized` and `volatile` constructs, as well as the `Thread.start()` and `Thread.join()` methods, can form *happens-before* relationships. In particular:

- Each action in a thread *happens-before* every action in that thread that comes later in the program's order.
- An unlock (`synchronized` block or method exit) of a monitor *happens-before* every subsequent lock (`synchronized` block or method entry) of that same monitor. And because the *happens-before* relation is transitive, all actions of a thread prior to unlocking *happen-before* all actions subsequent to any thread locking that monitor.
- A write to a `volatile` field *happens-before* every subsequent read of that same field. Writes and reads of `volatile` fields have similar memory consistency effects as entering and exiting monitors, but do *not* entail mutual exclusion locking.
- A call to start on a thread *happens-before* any action in the started thread.
- All actions in a thread *happen-before* any other thread successfully returns from a `join` on that thread.

The methods of all classes in `java.util.concurrent` and its subpackages extend these guarantees to higher-level synchronization. In particular:

- Actions in a thread prior to placing an object into any concurrent collection *happen-before* actions subsequent to the access or removal of that element from the collection in another thread.
- Actions in a thread prior to the submission of a `Runnable` to an `Executor` *happen-before* its execution begins. Similarly for `Callables` submitted to an `ExecutorService`.
- Actions taken by the asynchronous computation represented by a `Future` *happen-before* actions subsequent to the retrieval of the result via `Future.get()` in another thread.
- Actions prior to "releasing" synchronizer methods such as `Lock.unlock`, `Semaphore.release`, and `CountDownLatch.countDown` *happen-before* actions subsequent to a successful "acquiring" method such as `Lock.lock`, `Semaphore.acquire`, `Condition.await`, and `CountDownLatch.await` on the same synchronizer object in another thread.
- For each pair of threads that successfully exchange objects via an `Exchanger`, actions prior to the `exchange()` in each thread *happen-before* those subsequent to the corresponding `exchange()` in another thread.
- Actions prior to calling `CyclicBarrier.await` and `Phaser.awaitAdvance` (as well as its variants) *happen-before* actions performed by the barrier action, and actions performed by the barrier action *happen-before* actions subsequent to a successful return from the corresponding `await` in other threads.

# Data races defined in terms of happens-before

From the Java language specification (v. 15):

Two accesses to (reads of or writes to) the same variable are said to be conflicting if at least one of the accesses is a write.

[...]

When a program contains two conflicting accesses (§17.4.1) that are not ordered by a happens-before relationship, it is said to contain a data race.

[...]

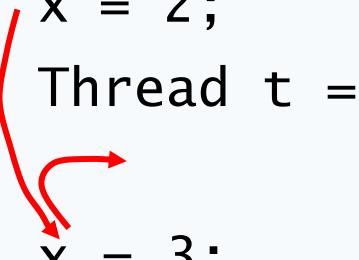
A program is correctly synchronized if and only if all sequentially consistent executions are free of data races.

[...]

If a program is correctly synchronized, then all executions of the program will appear to be sequentially consistent (§17.4.3).

# Happens-before example

```
static int x = 1;  
x = 2;  
Thread t = new Thread(() ->  
    System.out.println(x));  
x = 3;  
t.start();
```



- Data race because t reads x without synchronization?
- (Could argue read and write not overlapping in any SC execution.)
- x write *happens-before* x read,  
because *happens-before* transitive

SEARCH:  Search X

- they are guaranteed to traverse elements as they existed upon construction exactly once, and may (but are not guaranteed to) reflect any modifications subsequent to construction.

## Memory Consistency Properties

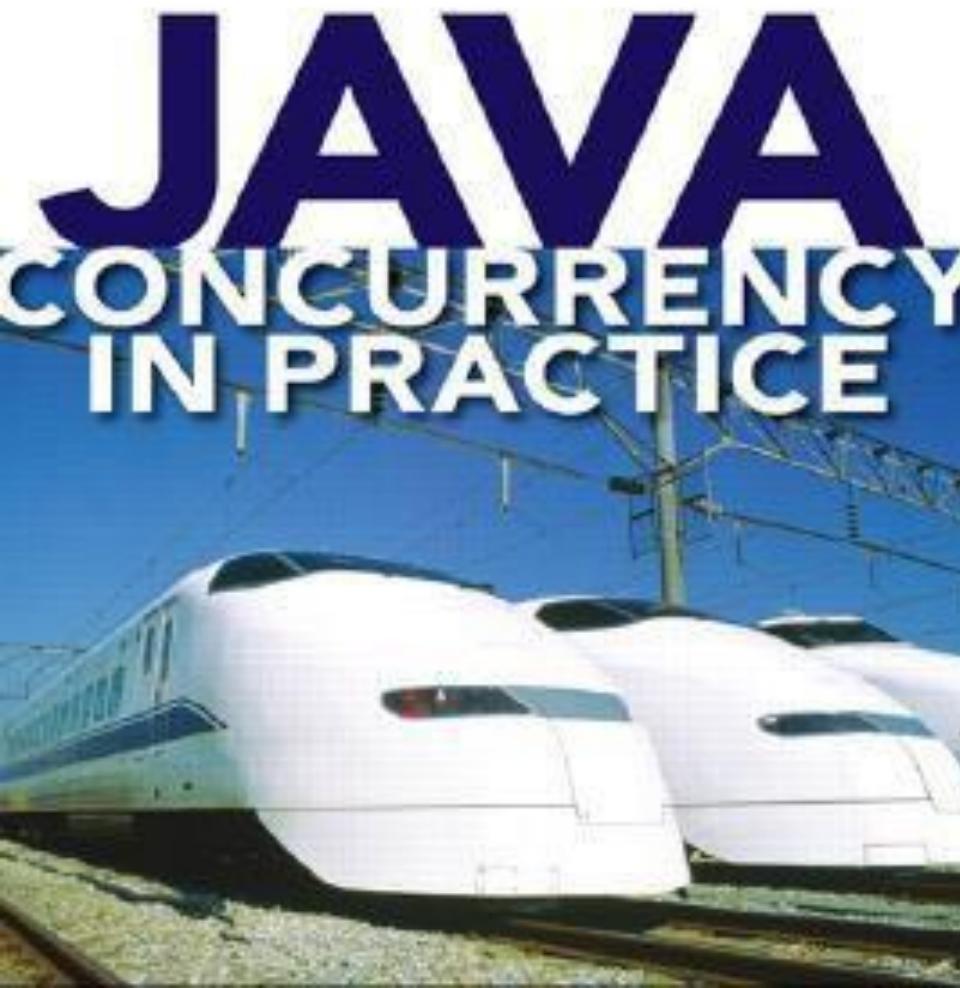
Chapter 17 of *The Java Language Specification* defines the *happens-before* relation on memory operations such as reads and writes of shared variables. The results of a write by one thread are guaranteed to be visible to a read by another thread only if the write operation *happens-before* the read operation. The `synchronized` and `volatile` constructs, as well as the `Thread.start()` and `Thread.join()` methods, can form *happens-before* relationships. In particular:

- Each action in a thread *happens-before* every action in that thread that comes later in the program's order.
  - An unlock (`synchronized` block or method exit) of a monitor *happens-before* every subsequent lock (`synchronized` block or method entry) of that same monitor. And because the *happens-before* relation is transitive, all actions of a thread prior to unlocking *happen-before* all actions subsequent to any thread locking that monitor.
- A write to a `volatile` field *happens-before* every subsequent read of that same field. Writes and reads of `volatile` fields have similar memory consistency effects as entering and exiting monitors, but do *not* entail mutual exclusion locking.
  - A call to `start` on a thread *happens-before* any action in the started thread.
  - All actions in a thread *happen-before* any other thread successfully returns from a `join` on that thread.

The methods of all classes in `java.util.concurrent` and its subpackages extend these guarantees to **higher-level synchronization**. In particular:

- Actions in a thread prior to placing an object into any concurrent collection *happen-before* actions subsequent to the access or removal of that element from the collection in another thread.
- Actions in a thread prior to the submission of a `Runnable` to an `Executor` *happen-before* its execution begins. Similarly for `Callable`s submitted to an `ExecutorService`.
- Actions taken by the asynchronous computation represented by a `Future` *happen-before* actions subsequent to the retrieval of the result via `Future.get()` in another thread.
- Actions prior to "releasing" synchronizer methods such as `Lock.unlock`, `Semaphore.release`, and `CountDownLatch.countDown` *happen-before* actions subsequent to a successful "acquiring" method such as `Lock.lock`, `Semaphore.acquire`, `Condition.await`, and `CountDownLatch.await` on the same synchronizer object in another thread.
- For each pair of threads that successfully exchange objects via an `Exchanger`, actions prior to the `exchange()` in each thread *happen-before* those subsequent to the corresponding `exchange()` in another thread.
- Actions prior to calling `CyclicBarrier.await` and `Phaser.awaitAdvance` (as well as its variants) *happen-before* actions performed by the barrier action, and actions performed by the barrier action *happen-before* actions subsequent to a successful return from the corresponding `await` in other threads.

# Demo OutOfOrderTest.java again



# Reading suggestions

- See *Java Concurrency in Practice* (2006) if you want more of this. The book presents simplified rules you can follow to do concurrent programming in Java instead of having to learn the details of the Java memory model.
- E.g., the book provides useful “safe publication idioms”
- Also e.g.: Hans-J. Boehm, “Threads cannot be implemented as a library” (2005).  
(<https://doi.org/10.1145/1065010.1065042>)
- Also e.g.: Hans-J. Boehm and Sarita V. Adve, “You don’t know jack about shared variables or memory models” (2012).  
(<https://doi.org/10.1145/2076450.2076465>)

# Advice from JCP, p. 16

- If multiple threads access the same mutable state variable without appropriate synchronization, *your program is broken*. There are three ways to fix it:
    - *Don't share* the state variable across threads;
    - Make the state variable *immutable*; or
    - Use *synchronization* whenever accessing the state variable.
- 
- Don't underestimate  
• the two first  
alternatives!

# Summary?

Make sure to not have data races in your Java programs

One way to think about all of this: Atomicity *and* **visibility**

Visibility aspect new in weak memory models compared to SC!